



Alice & Bob learn
**SECURE
CODING**

Tanya Janca
@shehackspurple

WILEY



Praise for *Alice & Bob Learn Secure Coding*

“Tanya’s book on secure coding is a brilliant example of what makes her a great expert and teacher. She takes complex material and makes it human, using clear, direct, and conversational language that sets it apart from most other books on similar topics. Her direct style shows that rather than trying to look smart, she’s actually teaching! The book is a welcome inhalation of pure knowledge.”

-Daniel Miessler, Founder of Unsupervised Learning

“Tanya Janca is a truly unique individual who has an incredible gift for turning complex and intimidating topics into something understandable and approachable for beginners. Her first book, *Alice and Bob Learn Application Security* is a perfect example of this. It’s one of the best resources out there, and I always recommend it to anyone who asks me how to get started with application security. Tanya’s impact goes beyond her writing. As a global speaker and instructor, she has empowered countless individuals to build more secure software and take their careers to new heights. I’m incredibly proud of her accomplishments and feel so lucky to have had her as a mentor! Tanya has been a huge inspiration in my career, and I’m constantly amazed by everything she’s achieving. I can’t wait to see what she does next—she’s truly making the world a safer, better place, one application at a time.”

-Rana Khalil, Distinguished Application Security
Professional and Trainer

"Tanya is a master at breaking down complex technical topics and making them both easily understandable and fun! I wish this book existed when I was first learning cybersecurity, as it's an excellent resource for security fundamentals and principles, important key tips for the most popular programming languages and frameworks, and how to follow a secure system development life cycle, along with many fun anecdotes and examples. Highly recommended for anyone who wants to rapidly learn a ton about secure coding from an industry veteran."

-**Clint Gibler**, Head of Security Research at Semgrep
and Founder of tl;dr sec

"Tanya has always been at the forefront of application security and practical learning. *Alice and Bob Learn Application Security*, her first book, was a great overview of application security practices, but this one is much more in-depth. Despite the subject being deeply technical, this book discusses it with fun and a different point of view while remaining relevant and specific to the subject. Only an author who masters the subject deeply is able to go so in-depth and keep it lighthearted."

-**Francesco Cipollone**, CEO & Founder @ Phoenix Security

"This book is hands-down one of the best resources out there for learning how to write secure code. The author has an incredible talent for breaking down tough security concepts and making them approachable without watering down the details. Each topic is presented in a way that feels thoughtful and intentional, and the examples are where the magic happens—they're clear, relatable, and, most importantly, actionable. These aren't just "nice-to-see" examples; they're the kind of scenarios you'll encounter in real projects, and they teach you exactly how to handle them securely.

What sets this book apart is its ability to cater to everyone, from beginners who are just getting their feet wet to experienced professionals looking to level up their skills. It doesn't just teach secure coding—it teaches you how to think about security as part of your coding process, which is invaluable in today's tech landscape."

-**Vandana Verma**, Security Relations Leader, Founder of
InfoSec Girls and InfoSec Kids, and OWASP BoD and Leader

"Alice and Bob Learn Secure Coding is almost as good as having Tanya in your office, chatting with you about application security concepts and details. You'll have a great time reading this book and will learn a lot along the way."

-Adam Shostack, Security Trainer, Author, Speaker,
and Threat Modeling Expert

"If you want simple, easy-to-follow guidance about secure coding from a verified authority on the subject, this book is for you."

-Ted Harrington, #1 Bestselling Author, Co-Founder of both
IoT Village and StartVRM, and Executive Partner at ISE

*"In all matters security, trust is earned, not given. In this book, Tanya solidifies the trust she earned in her first book, *Alice and Bob Learn Application Security*, this time as a source of secure coding wisdom and knowledge. Teams will be well served from learning the adventures of Alice and Bob as they journey toward more secure code!"*

-Izar Tarandach, Author of *Threat Modeling: A Practical Guide
for Development Teams*

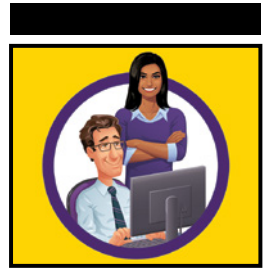
*"Want to stand out and take your software engineering career to the next level? You'll need to go beyond simply "making it work" and learn how to write high-quality and secure code. Fortunately, Tanya's unique skill and commitment to breaking down complex information without sacrificing rich, detailed technical content will make it easy for you to get started. This is a fantastic book for any software engineer to learn not just why, but *how* to write secure software, a skill that's much desired and highly valued in today's turbulent high-tech world."*

-Dustin Lehr, Co-Founder and CPTO of Katilyst Security,
Founder of Let's Talk Software Security, and Author of the
Security Champion Program Success Guide

“Tanya’s approach is refreshingly accessible and direct. She immediately addresses popular languages and frameworks before taking an in-depth approach to secure coding practices as they apply to each and every phase in the software development lifecycle.

This book is your authoritative guide to secure coding. Learn and enjoy!”

-Caroline Wong, Author and Cybersecurity
Expert Practitioner



Alice & Bob Learn Secure Coding

Tanya Janca

WILEY

Copyright © 2025 by John Wiley & Sons, Inc. All rights, including for text and data mining, AI training, and similar technologies, are reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Published simultaneously in Canada and the United Kingdom.

ISBNs: 9781394171705 (paperback), 9781394171729 (ePDF), 9781394171712 (ePub)

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at www.wiley.com/go/permission.

Trademarks: WILEY and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993. For product technical support, you can find answers to frequently asked questions or reach us via live chat at <https://support.wiley.com>.

If you believe you've found a mistake in this book, please bring it to our attention by emailing our reader support team at wileysupport@wiley.com with the subject line "Possible Book Errata Submission."

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Control Number: 2024951078

Cover images: © kbeis/Getty Images

Cover design: Wiley

*This book is dedicated to my favorite human in the entire world. Thank you for stopping the world for moments of time long enough for me to write this book.
With your support, I feel like I can do anything.*



About the Author

Tanya Janca, aka SheHacksPurple, is the best-selling author of *Alice and Bob Learn Secure Coding*, *Alice and Bob Learn Application Security*, and the *AppSec Antics* card game. Over her 28-year IT career, she has won countless awards (including OWASP Lifetime Distinguished Member and Hacker of the Year), has spoken all over the planet, and is a prolific blogger. Tanya has trained thousands of software developers and IT security professionals via her online academies (We Hack Purple and Semgrep Academy) and her live training programs. Having performed counterterrorism, led security for the 52nd Canadian general election, and developed or secured countless applications, Tanya Janca is widely considered an international authority on the security of software.



About the Technical Editors

Barre Dijkstra has spent over 25 years in software development in various roles, from hands-on development to management.

During this time, he started to place more and more emphasis on application security, from code to architecture and process. Nowadays, he's often supporting client teams on subjects like threat modeling, privacy, and security by design, as well as working with security teams from the development side.

Besides his work in software development, Barre is also a CTF player, works as a volunteer for the Dutch Institute for Vulnerability Disclosure (DIVD), co-organized one of the larger CTF meetups, and taught bachelor's-level courses in offensive security and reverse engineering for several semesters.

Fabien Lo Ricco is a systems engineer with 15 years of comprehensive experience in software engineering, systems engineering, and technical support. Specializing in security engineering with a focus on data loss prevention (DLP), he brings hands-on expertise in protecting sensitive information.

Throughout his career serving small businesses, he has developed a versatile skillset that spans the entire IT spectrum—from helpdesk support to systems architecture. He is a strong advocate for open source software and leverages both proprietary and open source solutions to design robust and cost-effective systems tailored to business needs.

Franziska Passing is a senior operations engineer at Duolingo. She has previously worked in both platform engineering and security engineering at companies such as Mapbox, Microsoft, and others. She is an experienced conference speaker, tech community organizer, and active member of the feminist hackspace Heart of Code. Currently residing in Berlin, she enjoys drinking coffee and participating in triathlons. You can contact her through her website, <https://franka.tech>.



Acknowledgments

I would like to express my deepest gratitude to my three outstanding technical editors (Fabien Lo Ricco, Franka Schmidt, and Barre Dijkstra), whose unwavering support, constructive feedback, and dedication have been instrumental throughout the creation of my second book. Their encouragement, perseverance, and commitment to excellence have not only strengthened the quality of the work but also inspired me to reach new heights in my writing journey. I am truly indebted to them for their invaluable contributions and for standing by me for over two years, navigating the ups and downs of this book-writing process. I cannot thank them enough for their guidance, expertise, and belief in this project.



Contents at a Glance

Foreword		xxvii
Introduction		xxix
Part I	General Advice	1
Chapter 1	Introductory Security Fundamentals	3
Chapter 2	Beginning	21
Chapter 3	Improving	55
Chapter 4	Achieving	81
	Summary of Part I	117
Part II	Specific Advice	125
Chapter 5	Technology-Specific	127
Chapter 6	Popular Programming Languages	143
Chapter 7	Popular Frameworks	181
Chapter 8	Vulnerability Categories	211
	Summary of Part II	245
Part III	Secure System Development Life Cycle	251
Chapter 9	Requirements	253
Chapter 10	Design	259
Chapter 11	Coding	267
Chapter 12	Testing	279

Chapter 13	Release/Deployment	295
Chapter 14	Maintenance	305
Chapter 15	Conclusion	319
	Summary of Part III	339
Appendix A	Resources	343
Appendix B	Answer Keys	361
Index		377



Contents

Foreword	xxvii
Introduction	xxix
Part I General Advice	1
Chapter 1 Introductory Security Fundamentals	3
Assume All Other Systems and Data Are Insecure	3
The CIA Triad	4
Least Privilege	6
Secure Defaults/Paved Roads	8
Assume Breach / Plan For Failure	9
Zero Trust	9
Defense in Depth	10
Supply Chain Security	10
Security by Obscurity	11
Attack Surface Reduction	11
Usable Security	12
Fail Closed/Safe, Then Roll Back	12
Compliance, Laws, and Regulations	12
Security Frameworks	14
Learning from Mistakes and Sharing Those Lessons	16
Backward Compatibility (and Potential Risks It Introduces)	16
Threat Modeling	16
The Difficulty of Patching	17
Retesting Fixes for New Security Bugs	18
Chapter Exercises	19

Chapter 2	Beginning	21
	Follow a Secure System Development Life Cycle	21
	Use a Modern Framework and All Available	
	Security Features Within	22
	Input Validation	23
	Output Encoding	26
	Examples of Output Encoding	27
	HTML Context	28
	JavaScript Context	28
	Parameterized Queries and ORMs	29
	Authentication and Identity	31
	Authorization and Access Control	32
	Access Control Models	33
	Logical Access Control Methods (Implementation)	34
	Session Management	34
	Secret Management	35
	Password Management	37
	Communication Security (Cryptography and	
	HTTPS Only)	39
	Protecting Sensitive Data	40
	Security Headers	43
	New Security Header Features	43
	Fetch Metadata Request Headers	43
	Content Security Policy Header	44
	Strict-Dynamic	44
	Trusted-Types	44
	Security Headers Previously Covered	44
	Content-Security-Policy Header	45
	HTTP Strict-Transport-Security	45
	X-Frame-Options	45
	X-Content-Type-Options	45
	Permissions Policy	46
	Expect-CT	46
	Referrer-Policy	46
	Public Key Pinning Extension for HTTP (HPKP)	46
	X-XSS-Protection	46
	More New Headers	46
	Same-Origin Policy	47
	COEP: Cross-Origin Embedder Policy	47
	COOP: Cross-Origin Opener Policy	48
	CORP: Cross-Origin Resource Policy	48
	CORS: Cross-Origin Resource Sharing	48
	CORB: Cross-Origin Read Blocking	49
	Secure Cookies	50
	Error Handling	51
	Chapter Exercises	52

Chapter 3	Improving	55
	Database Security	56
	Four Perspectives for Protecting Databases	56
	File Management	59
	File Uploads	61
	Your Source Code	62
	Memory Management (Buffer, Stack, String, and Integer Overflows)	63
	How Do We Avoid Overflows?	64
	(De)Serialization	66
	Privacy (User/Citizen/Customer/Employee)	67
	Errors	69
	Logging, Monitoring, and Alerting	72
	Fail Closed	73
	Locking Resources	73
	Enabling Password Managers	74
	Cryptographic Practices	75
	Strongly Typed Languages	76
	Strongly Typed Languages	76
	Weakly Typed Programming Languages	77
	Domain-Driven Development	78
	Memory-Safe Languages	79
	Chapter Exercises	80
Chapter 4	Achieving	81
	Secure Design	82
	<i>How much is “enough” (design) security?</i>	84
	<i>Dependency Management and Supply Chain Security</i>	85
	Dependency Security	86
	Checking If Dependencies Are Safe to Use	87
	Supply Chain Security	87
	<i>Secure Defaults</i>	90
	Secure Defaults for Users	90
	Secure Defaults for Developers	92
	<i>Readable and Auditable Code</i>	93
	<i>Important Functions Happen on Trusted Systems</i>	96
	What Is an “Untrusted” System?	96
	What Are “Important Functions”?	97
	Putting It Together	97
	<i>Allowlists versus Blocklists</i>	97
	Why Are Block Lists Bad?	98
	How Do We Create an Allowlist?	98
	<i>Secure Configurations</i>	99
	<i>Hostname Validation</i>	100
	<i>Reusable Code</i>	100

	<i>Safe System Calls</i>	102
	Mitigating Circumstances	102
	<i>Commenting and Other Documentation</i>	102
	Comments	103
	Documentation	104
	<i>Verification of User Consent</i>	106
	<i>Integrity Checks, Code Signing, and Immutable Builds</i>	107
	Immutable Builds	108
	<i>Avoiding Brute Force</i>	109
	<i>Security Controls</i>	110
	<i>Handling Elevated Privileges</i>	111
	<i>Security Maintenance</i>	112
	<i>Repaying Technical Debt</i>	113
	Chapter Exercises	114
	Summary of Part I	117
	Checklist of General Secure Coding Advice	117
Part II	Specific Advice	125
Chapter 5	Technology-Specific	127
	API Security Best Practices	127
	Mobile Application Security Best Practices	134
	WebSocket Security Best Practices	137
	Serverless Security Best Practices	138
	IoT Security Best Practices	140
	Chapter Exercises	141
Chapter 6	Popular Programming Languages	143
	JavaScript	143
	HTML/CSS	148
	HTML5, Specifically	149
	Python	151
	SQL	154
	Node.js	157
	Java	160
	Serialization in Java	164
	TypeScript	165
	C#	166
	PHP	170
	C/C++	175
	Conclusion	178
	Chapter Exercises	179
Chapter 7	Popular Frameworks	181
	Web and JavaScript	181
	Express	182
	React.js	184

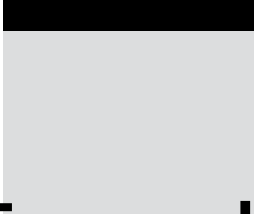
Angular	186
jQuery	190
Vue.js	192
Other Frameworks and Libraries	194
.NET (Core)	194
Ruby on Rails	199
Spring and Spring Boot	204
Flask	207
Chapter Exercises	210
Chapter 8 Vulnerability Categories	211
Design Flaws / Logic Flaws	212
How Does This Happen?	213
The Risk	213
Prevention	214
Code Bugs / Implementation Errors	215
How Does This Happen?	215
The Risk	215
Prevention	215
Overflows and Other Memory Issues	216
Overflows	216
Buffer Overreads	217
Invalid Page Faults	217
Use After Free	218
Uninitialized Variables	218
Memory Leaks	218
How Does This Happen?	219
The Risk	219
Prevention	219
Injection: Interpreter and Compiler Issues	220
How Does This Happen?	221
The Risk	221
Prevention	221
Input Issues	222
How Does This Happen?	223
The Risk	223
Prevention	223
Authentication and Identity Issues	223
How Does This Happen?	224
The Risk	224
Prevention	224
Authorization and Access Issues	225
How Does This Happen?	225
Configuration and Implementation Issues	225
How Does This Happen?	226
The Risk	226
Prevention	226

Fraudulent Transactions	227
How Does This Happen?	227
The Risk	227
Prevention	228
Replay Attacks	228
How Does This Happen?	228
The Risk	229
Prevention	229
Crossing Trust Boundaries	229
How Does This Happen?	230
The Risk	230
Prevention	230
File Handling Issues	230
How Does This Happen?	231
The Risk	231
Prevention	231
Object Handling Issues	232
Prominent Features of OOP	232
Deserialization and Other Object Handling Issues	234
How Does This Happen?	234
The Risk	234
Prevention	234
Secrets Management Issues	235
How Does This Happen?	236
The Risk	236
Prevention	236
Race Conditions and Timing Issues	237
How Does This Happen?	237
The Risk	238
Prevention	238
Resource Issues	240
How Does This Happen?	240
The Risk	241
Prevention	241
Falling into an Unknown State	241
How Does This Happen?	242
The Risk	242
Prevention	242
Chapter Exercises	243
Summary of Part II	245
Checklist of Technology-Specific Secure	
Coding Advice	245
Checklist of Secure Coding Advice for	
Languages and Frameworks	246
Summary of Vulnerability Issues to Watch For	248

Part III	Secure System Development Life Cycle	251
Chapter 9	Requirements	253
	Project Kick-Off: Outline of Your Project's Security Activities	253
	Project Scheduling and Planning	254
	Security Requirements	255
	Chapter Exercises	257
Chapter 10	Design	259
	Threat Modeling	260
	Secure Design Patterns and Concepts	262
	Architecture Whiteboarding	263
	Examining Data Flows	263
	Security User Stories	264
	Chapter Exercises	265
Chapter 11	Coding	267
	Training	267
	Organizations	269
	Individuals	270
	Code Review	270
	First- and Second-Generation Static Analysis Tools	271
	Secure Guardrails	272
	IDE Plugins and Other Guidance	273
	Verifying That Your Dependencies Are Safe (SCA)	274
	How Do You Decide Which Dependencies Are Worth Updating or Changing?	274
	Finding and Managing Secrets	275
	Dynamic Testing (DAST)	276
	Chapter Exercises	278
Chapter 12	Testing	279
	Test Coverage and Timing	280
	Depth Versus Coverage	281
	Scanning Your Infrastructure	281
	Production or Lower-Level Environments	281
	Scoping	282
	Timing	282
	Manual Testing	284
	Automated Testing	286
	Fuzzing	287
	Interactive Application Security Testing (IAST)	288
	Bug Bounty Programs	289
	Test Results	290
	Actioning Test Results	291
	Final Thoughts	293
	Chapter Exercises	293

Chapter 13	Release/Deployment	295
	Security Events Within the CI/CD	296
	Breaking the Build	297
	Secret Scanning	298
	Static Analysis	298
	Dynamic Analysis	298
	Software Composition Analysis	299
	Linting	299
	Infrastructure as Code scanners	299
	Securing the CI/CD Pipeline Itself	299
	Assuring the Integrity of Your Release	302
	Security Release Approval	303
	Chapter Exercises	304
Chapter 14	Maintenance	305
	Monitoring, Alerting, and Observability	306
	Blocking/Shielding	308
	Web Application Firewalls (WAFs)	309
	Content Delivery Networks (CDNs)	309
	Runtime Application Self-Protection (RASP)	310
	Virtual Patching	310
	API Gateways	310
	A Special Note for Data Scientists	311
	Continuous Testing	312
	Security Incidents	313
	Business Continuity and Disaster Recovery Planning	315
	Chapter Exercises	317
Chapter 15	Conclusion	319
	Good Habits	319
	Your Responsibility	322
	How Much Is <i>Enough</i> ?	323
	Using Artificial Intelligence <i>Safely</i>	325
	Continuous Learning	327
	Becoming a Champion	328
	Getting Others on Board	330
	Transitioning onto the Security Team	330
	Applying for Security Jobs Outside of Your Organization	331
	Conclusion	335
	Summary of Part III	339
	Checklist of Security Activities for Each Phase of the SDLC	339
Appendix A	Resources	343
	Chapter 1: Introductory Security Fundamentals	343
	Chapter 2: Beginning	344
	Chapter 3: Improving	345
	Chapter 4: Achieving	347

Chapter 5: Technology-Specific	349
Chapter 6: Popular Programming Languages	351
Chapter 7: Popular Frameworks	355
Chapter 8: Vulnerability Categories	357
Chapter 10: Design	359
Chapter 11: Coding	359
Chapter 12: Testing	359
Chapter 13: Release/Deployment	360
Chapter 14: Maintenance	360
Appendix B Answer Keys	361
Chapter 1: Introductory Security Fundamentals	361
Chapter 2: Beginning	363
Chapter 3: Improving	364
Chapter 4: Achieving	365
Chapter 5: Technology-Specific	368
Chapter 8: Vulnerability Categories	370
Chapter 9: Requirements	371
Chapter 11: Coding	372
Chapter 12: Testing	373
Chapter 13: Release/Deployment	374
Chapter 14: Maintenance	375
Index	377



Foreword

From a technological point of view, it's a pretty amazing time to be a software engineer. When I was a much younger nerd, reading science fiction novels and watching early episodes of *Doctor Who* and *Star Trek: The Next Generation*, I was fascinated by the technologies on display. We had applications and devices to solve every problem in this new future.

Fast-forward 25 years, and it feels like we are living in the age of extraordinary invention. Software touches almost every part of our lives, from our work to our free time, and we, as engineers, are the ones building it.

The systems we write help bring us together and help us collaborate and share at a scale never before seen. Software is also helping us identify medical conditions earlier, control robots that can help us navigate unsafe environments, and even drive around our cities for us. With this great focus on software and the benefits it can bring us, however, comes the responsibility we share for what this software can do and its impacts on people, data, and systems.

When I started my software development career, the systems I built (in the finance and taxation space) were less crucial than they are now. The systems I built were broadly only used between 9:00 a.m. and 5:00 p.m., Monday to Friday, and if they stopped working, although it was annoying, many manual systems could compensate.

The software we build now is integrated so deeply into our lives that the impacts of incidents and issues are much more varied and, in some cases, much more dangerous.

That is why this book and Tanya's lifetime of work in secure coding are so crucial. As software developers, we are at a crossroads. We have an obligation to make our software and systems of the highest quality and standards.

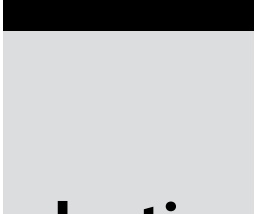
We have a responsibility to balance performance, scaling, usability, observability, and accessibility—ensuring that our applications do the job as intended, when needed, and for as many people as possible.

Now, it's time to add security to this mix.

In this book, Tanya has made secure coding accessible to software developers from all walks of life and all software ecosystems. Her easy-to-understand examples and simple, pragmatic approach mean that even the busiest software teams can get started securing their code in a way that works for their software development life cycle, culture, and language choices. Packed with examples and sidebars, this book is easily the grab-and-go reference you want to give your team.

If we are going to build this wonderful software-centric future together, I hope we can all commit to making every day a little more security-focused than the last and every line of code an opportunity to improve our software quality and security.

Laura Main Bell
CEO and Founder, SafeStack



Introduction

Right now, our industry demands more from programmers than ever before; I'm sure you've all seen articles urging you to be a 10-X engineer, be an expert in several programming languages, and conquer all aspects of full-stack development. But one clear and unnegotiable expectation stands out for software developers: you must create secure applications. It doesn't matter if it wasn't covered in your computer science program, if your security team is unhelpful, or if you don't have the right toolset available. Your users, teammates, and the entire organization are counting on you to provide them with reliable and safe software.

This book will help you write more secure code, full stop. It is a toolkit for standing out among your peers, developing robust applications, and safeguarding their data. By reading this book, you are taking a giant step toward becoming a senior software engineer or the newest member of the security team.

Unlike many other textbooks, this one will make you smile. You may also find it surprisingly easy to understand and consume. As a previous professional entertainer who also has dyslexia, I focused on making this book a joy to read and as easy as possible to understand. Using both humor and numerous teaching tactics (analogies, relatable stories, empathy for Alice and Bob, diagrams, repeating complex abstract concepts, etc.), I tried to write a book that I would have wanted earlier in my career.

This book is a whirlwind of secure coding, covering advice that applies to every tech stack, as well as more specific and detailed advice for 10 programming languages, 9 frameworks, several technologies, and every vulnerability class I am aware of. That is *a lot* of information. It is easily enough material to create an amazing secure coding guideline for your workplace. Wink, wink.

This isn't just a book; it's an interactive journey. Over the course of 2025 and 2026, I will be doing monthly livestreams for each chapter of the book. These sessions will feature my (expert) friends and me discussing each chapter, including the answers at the end, plus live Q&A.

I didn't just write this book for myself; I wrote it for you. When I started to learn about security, it was a *struggle*. The few resources I could find were not very good. I made it my personal mission to help streamline the transition to application security and improve our industry's approach to software security. But as time passed, I saw that if I wanted to create substantial change, I needed to turn my attention to developers. And so here I am, dear reader.

I wrote this book for one more reason, and I suspect it's the same reason you picked it up. I want myself and those I love to be able to use software *safely*. I want the products our industry creates to be *trustworthy*. As of this writing, that is not the case; we've all heard of endless data breaches, security incidents, and other damage or injury happening as a result of insecure applications. While we wait for our governments to get their collective acts together, I am trying to raise the bar. With this book, I hope to instill a sense of responsibility in every developer to build more secure software.

Everyone is driven by something different in their lives. For me personally, it is extremely important that I perform good in my life. By sharing knowledge and cultivating a community of developers committed to building secure software, I hope to protect countless individuals from harm. But for this to happen, I need your commitment. Are you ready to go on this journey with me?

I invite you to immerse yourself in this book and use the knowledge within to create truly remarkable software.

Part



General Advice

In This Part

Chapter 1: Introductory Security Fundamentals

Chapter 2: Beginning

Chapter 3: Improving

Chapter 4: Achieving

Introductory Security Fundamentals

This chapter will focus on fundamental security concepts, the sorts of things that every security person wishes every other IT professional knew. You may not have a chance to apply all of them in your work, but understanding them can help you create more robust and defensible systems.

Assume All Other Systems and Data Are Insecure

Perhaps the most important lesson in this entire book is learning to stop trusting computer systems, data, and users by default. Human beings, in general, are quite trusting as a species.¹ For creating a society with laws, safety, and general order, having most people assume trust from the start is a good thing. It's part of what makes our societal fabric work.

As a result of human beings generally assuming trust, when we design computer systems, we tend to design them in such a way that the systems have an *implied trust*. What this means is that rather than automatically verifying facts, our computer systems assume they must be true. And this can lead to dire consequences.

COLLEGE COPY CAT

When Alice was in college, she used to save her work files to a shared drive on the network for school. In case her computer ever failed, she knew there would be a backup. In her second year of college, she was preparing a presentation for her software engineering course as part of a class team project. During the class, when her team was going to present the project research, a classmate named Eve² asked if she could present her own project research first. Alice didn't see anything wrong with this, so she agreed. Eve got up and presented all of Alice's findings as though they were her own! She even used Alice's slide deck and changed the name on the front page to her own. Alice thought, "*Are you kidding? How could this happen!?!?!?*" All of their classmates applauded and were very pleased with Eve. Alice had assumed that saving her files on a shared drive at school meant they would be safe. She never thought a classmate would steal her work and present it as their own or that someone would go rifling through Alice's folders on the shared school drive. Alice had *assumed trust*, and she had gotten burned.

If we are trying to create a secure system, it is of the utmost importance that we never assume trust. This can mean using multifactor authentication to protect against credential stuffing attacks, double-checking data you receive from the database to ensure that it is the correct format and size, or performing authentication and authorization against an application programming interface (API) calling a serverless app, even though they were both built by your company.

Examples of implied trust:

- Zoning in network design: once someone has entered a zone, they are able to access every other system within that zone without having to cross a firewall or reauthenticate.
- Accepting user input, not validating it, and then using that input to create an SQL query, a URL redirect, or another decision within your system.
- Exposing an API to the internet without putting a gateway in front of it or any other mechanism to perform authentication and authorization, allowing anyone to call it—including bots.

If you only learn one thing from this book, I hope it is this: design every system with as little implied trust as possible. Verify everything, including input, decisions, data, and other system integrations. Perform more than one verification if possible and several if the stakes are high (top-secret information, systems requiring high availability, etc.). Always assume that other systems and data are potentially insecure.

The CIA Triad

The CIA triad are the three things that make up the mandate for most IT Security teams around the world. It is a cybersecurity team's job to guard the confidentiality,

integrity, and availability of the systems and data they are charged with protecting. Security teams protect more than just the CIA, but those are often considered the core topics. Throughout this book, we will cover much more, including privacy, safety, authenticity, and layering our defenses.

TOP-SECRET WORK

Bob used to work on a top-secret case for the government of Canada. It was about antiterrorism activities, and that's all he was allowed to say about it. Keeping that sensitive data safe was of the utmost importance, and Bob took all of his training on how to protect that data very seriously. When Bob had his training with the Canadian Centre for Cyber Security, he asked what made the data "top secret" rather than secret or some other classification. The response was, "If that data got out, it could potentially harm Canada as a nation. It could potentially result in hundreds of lost lives and cause various other counterterrorism activities to fail. This information could not only harm those working for the government but also block them from uncovering various plots that could lead to the death of citizens or even, in the worst case, a successful government coup. Your most important task, no matter how complex, onerous, or difficult, is to keep this information from falling into the wrong hands. Guard it, literally, with your life." Although Bob rarely spoke about this work assignment with his friends and family, they were all very aware of how Bob felt about the importance of his work for the Department of Justice.

NOTE

- Confidentiality: the state of keeping or being kept secret or private³
- Integrity: internal consistency or lack of corruption in electronic data³
- Availability: the quality of being able to be used or obtained at any time³

WHEN CRITICAL INFRASTRUCTURE GOES DOWN

In 2022, on July 9, the Rogers telecommunications network went down in Canada.⁴ Canada has only three major telecommunication companies, and the rest share or rent the lines from the main three. Canadians couldn't pay bills, surf the net, or even call 911 in large parts of the country for several days as a result of this outage. Cybersecurity generally focuses on three things: confidentiality, integrity, and availability (CIA). As you can see, when Rogers went down, it caused a lack of *availability* for much of the country, and therefore the outage was a *security issue*. If one human error can disable emergency services for a large part of the country for several days, that system *is not secure*. Although the outage itself was not caused by a cyber attack, it interfered with one of the CIA triad, and therefore was a security issue. The lack of contingency planning around these systems is *also* a security issue. This note is not meant as a critique against Rogers; it is a real-world example of how security is *a part of quality* and also the importance of defense in depth, business continuity planning, and disaster recovery.

Although we have seen the definition of *integrity*, let's talk a little more about the meaning of that word. Often, when we speak of a person who "has integrity," we mean that you can trust that person: you can rely on them and know they will always make the "right" decision. Integrity is similar for computer systems; when a system and its data have integrity, it means we can use that data to make decisions and know they will be good decisions. This data has already been verified; it is trustworthy.

Now imagine that you are a doctor, and you use a computer to calculate a dosage for medicine based on your patient's weight, height, any medical conditions they have, and various other factors. When you give the patient the medication, you *assume* it's the correct amount—that the information the computer gives you will help the patient, not harm them. When you do this (trust the output of the system), you are assuming that the system and its data *have integrity*. Imagine the horror for a doctor if the computer got it wrong and their patient was caused harm. For some systems, the integrity of the data is critical (measuring medicine, for instance), and for others, it's not so important (a recipe for a cake that says one egg versus two—it's not the end of the world). When you are creating a system, it's extremely helpful to know which of the CIA is most important and then to design your systems and tests with that in mind.

Least Privilege

The principle of least privilege (PoLP) refers to an information security concept in which a user is given the minimum levels of access – or permissions – needed to perform their job functions.

— CyberArk⁵

Although this quote implies that least privilege only applies to users, this is not true; it applies to any person or thing who may have access or privileges, including computer systems, like software or an AI. Least privilege also applies to how long the access is given, as it should only be provided during the time it is required and no longer.

NO MORE ADMIN RIGHTS

Alice remembers her first introduction to the concept of least privilege from the security team. They told her she couldn't have admin privileges on her work desktop anymore. Alice was not pleased. She tried to explain that as a C-level exec, she *needed* to have admin privileges so that she could install software she found on the internet and not have to wait on tech support's approval. The security team informed her not

only that she should not be installing software that was not provided by the company but also that this was part of the reason they were taking this privilege away. They told her several executives had been phished, and because they had admin privileges and no anti-malware software, it caused a lot of problems. They told her that as an executive, she was a target for malicious actors because she had so much power within the company. They also explained that she would definitely still be able to do her job without it, and if she had a need for software they were not supplying, they would get it for her—just ask. Alice was not exactly pleased with the conversation, but it seemed clear these security folks were not going to back down, so she said, “Okay, fine!” (as though she had any choice in the matter).

Whenever we publish or deploy a new piece of software or add a user to a network or any other electronic system, we have to decide how much access they will have and for how long. If we give them access to everything, this means they have the ability to cause damage to everything within that system. This could be intentional or accidental. It could also be that a regular user’s account was taken over by a malicious actor and then used to wreak havoc on your network, application, or anything else they have been given access to.

That said, if we limit their rights and privileges by only giving them the access that they require to perform their job function, the amount of damage someone could do would be greatly reduced. This applies to all electronic systems, including software! Let’s walk through some examples.

Example 1: Walter is a parent of teenage twins, and their curfew is midnight. Walter has electronic smart locks on his home that deny entrance for their fingerprints starting at 12:05 a.m. This means if his teenagers stay out past curfew, they have to ring the doorbell, alerting both parents that they arrived home late. However, the parents can enter the home at any time. The parent’s privileges to access their home are different, and the twins have fewer privileges. Walter quite likes it that way!

Example 2: You build a graphical user interface (GUI) front end for your application, plus three APIs that perform services for your front end and one database to hold all the data for you. Your application has one service account created on your network; this account is your application’s identity on the network. Ideally, your APIs can only be called by the corresponding service account, and all other calls will be blocked or ignored. Your database should only accept connections from your APIs that belong to your app, using your app’s service account, and from the database administrator (DBA). This way, if another app or network entity tries to send commands to your database or make calls to your APIs, it won’t work. You could protect your system’s data further by selecting the least amount of privileges to your data that you need: for instance, selecting read-only or create-read-update-delete (CRUD) rather than database owner (DBO).

Example 3: You are in charge of access to a lab where sensitive research is being performed. You give everyone their own access card, with a photo of them, that needs to be scanned to get into each of the different laboratories. You set access to various closets or cabinets with the same system but only let people into the areas they are supposed to have access to. For instance, some scientists are allowed to access chemicals, whereas others may use the very special equipment for measuring experiments. Each employee can only access what they need to do their jobs, and nothing more. When employees stop working for the company or stop performing the role that requires this access, the access is revoked.

Example 3, expanded: The laboratories you are protecting have extremely sensitive and dangerous chemicals, and there are top-secret experiments being performed. The risks in this situation are both physical and political, and thus you increase the level of security and reduce the amount of privileges each person is given. To allow someone into the building, they must swipe their badge and have their body and belongings scanned for weapons; then they enter a cage where they are visually identified by one of the security guards. Once they are in, certain cabinets are only opened with two people's access cards, not one. The important cabinets require multiparty authorization.

Secure Defaults/Paved Roads

Every computer system that has any configuration has defaults; they are the settings that come with the system by default. Often, users leave whatever the default settings are for their entire usage of the software. This means if the original setting on your software is suboptimal with regard to privacy or security, most of your users will continue to use it in this state. Many software creators take advantage of this, using it to violate their users' privacy by selling their data, sharing it with other organizations, and other less-than-ideal usage.

SECRET SIGNALS

Alice used to use Telegram to send messages to friends because she thought it was very secure and had end-to-end (e2e) encryption. One day a friend told her that it's only encrypted from the client to the server, but on the server, it's decrypted.⁶ All those years, she thought her messages had been encrypted and safe, even from the people at Telegram! She felt like an idiot and switched to Signal Messenger.

That said, we security-minded IT professionals can take advantage of this as well, by setting defaults that are secure. When we make the path of least resistance the secure path, everyone wins. If it is possible to make the easiest way to do something also the most secure way to do it, we are more likely to get the

results we want! Whenever possible, create a paved road that leads to the most secure way forward for the user.

Assume Breach / Plan For Failure

The concept of *assume breach* means to plan, design, and react as though we have been breached, or will be one day. A breach can be someone breaking into a system, stealing your data, breaking into a building, and so on.

Examples of this would be launching your incident response process the moment a big vulnerability is reported to your bug bounty program or your coordinated disclosure program, using multiple layers of defense in case one is surpassed, or monitoring inside your network for suspicious activity.

Planning for failure is very similar to “assume breach,” except that it does not include reacting to events, just planning for them. Designing systems to have multiple layers of defense, including contingency plans, and ensuring that both our business continuity plan (BCP) and our disaster recovery (DR) plans are up to date and actionable are some of the ways that we, as IT professionals, can plan for failure.

REAL-LIFE PLANNING FOR FAILURE

A real-life example of planning for failure is the COVID-19 pandemic hitting its peak in 2020. Think back to when organizations quickly switched to work-from-home, selling their products online, or deploying systems to ensure that employees were safe. Some organizations were ahead of the curve, and it is highly likely that they enacted their BCPs to achieve this. Although data is not publicly available, I (the author) would love to see the data on the financial cost for organizations that did and did not have BCPs and DR plans and how much money, time, and human lives were saved as a result of planning for failure.

Zero Trust

Zero Trust is a concept that is the opposite of assumed or implied trust. If you recall the first section in this chapter, “Assume All Other Systems and Data Are Insecure,” this is an extension of that concept. Assumed trust means that after initial verification, we trust a system, person, or account, usually permanently. For instance, someone logging in to a network is allowed to access anything on the whole network, or a person going into a building is allowed to go into every single room on every floor.

In real life, humans are generally trusting, but it depends on the social situation. Imagine having a guest over for a dinner party in your home and finding them

going through your medicine cabinet: that would be weird. It would set off alarm bells. You would likely ask them what they are doing or even kick them out of your home. But computers don't understand social situations, so we have to teach them in advance.

Zero Trust means that there is no implied or assumed trust, ever. All systems deny access as a default. Unless a human gave access explicitly, the answer is no. When we implement Zero Trust, this means we have to be careful to ensure that everything is set up properly or things break. It's a labor-intensive concept to apply, but when done properly, it is extremely effective.

Defense in Depth

Defense in Depth is the simplest of all the security concepts; it means having more than one layer of defense. Some people have a lock on their front door, which is one layer of defense. Others have a lock, an alarm system, video cameras, and an attack dog. That is defense in depth: multiple defenses that are layered. We do this because there are often weaknesses or unknown chinks in our armor; multiple layers can ensure that attackers are blocked. Ideally, you will decide how many layers of defense are required based on the value of the system and/or data you are protecting and the risk of said systems and/or data coming to harm.

Supply Chain Security

A *supply chain* is all the things you need to create a product. Let's say you want to make tomato sauce to sell at grocery stores. You will need tomatoes, spices, salt, (possibly) sugar, and a jar to hold it all. You may also want to put a nice label on the jar, meaning you need the label and some glue. To create the product regularly, you will need to have a regular supply of all of these things, and several of them need to be fresh (rotten tomatoes would not make a tasty sauce).

The label, jar, and glue are all products in themselves, which means each of them also has its own supply chain. Each one of their ingredients is part of your tomato sauce supply chain. As you list each and every item and then chain them together in a list, that is the *supply chain* for your product.

Supply chain security means protecting the entire chain of ingredients. Personally, I don't even know what the ingredients are for glue, but I do know that without it, the labels will fall off. Every single part of the supply chain needs to be kept safe and protected; we can't have broken glass jars or poisonous sauce.

With software, this means each one of the dependencies you have included in your application has to be safe to use. If those dependencies call other libraries,

packages, or software, all of that needs to be secure as well! It also includes every tool you use to create the software, such as your Integrated Development Environment (IDE), testing servers, your version control system, and so on. We will talk later in the book about how to do a good job of this, but for now, just understanding that every component, library, framework, or package you use to build your software is part of your supply chain is enough.

Security by Obscurity

This concept is mostly referred to with regard to open-source code versus proprietary code, and which is more secure. It is my opinion that all software can have bugs and flaws, and it is the systems we put in place to detect, fix, and avoid security issues that determine the security of its end state. That said, not sharing a copy of our code publicly would make it more difficult for a determined attacker to analyze it for vulnerabilities. Not broadcasting the SSID (name) of our Wi-Fi home network can help us avoid random members of the public trying to brute force their way onto it, but a determined or advanced attacker will generally find it. Although “security by obscurity” should never be used as the *only* defense of a system, it can help protect intellectual property and be used as one of several layers in a more comprehensive security plan.

Attack Surface Reduction

Every part of your application that someone can interact with is part of your app that can be attacked. Anything that can be attacked is considered part of your *attack surface*. If we can reduce the amount of attack surface we have, there is less for malicious actors to try to wreak havoc on, meaning there is less chance they will be successful.

One of the things that is often unstated about reducing our attack surface is that unused code is usually what we end up removing, and unused code is often insecure code! For instance, if there’s a menu item for a feature no one uses anymore, you should *remove the code from your application*, not just comment out the menu item. Over time, this unused feature will likely not be tested and updated because it’s not on the list of things for the QA team and penetration tester to look at. It might get missed, and if it does, that means it’s less likely to be secure. Another bonus when reducing the size of your codebase is that it will make it easier to understand and maintain.

NOTE Key takeaway: Unused code is significantly more likely to be insecure, and thus we should remove it from our apps.

Usable Security

Human beings will always find a way to get their jobs done, we are a determined bunch. When the security team puts a security control in place that prevents them from performing their duties, even the least technical of people will find a way around it; it's human nature. When users find a workaround, it is often a less secure way to use the system. It is best if the security team works hard to ensure that the security mitigations they put in place don't cause difficulty for the end users. Asking for feedback and adjusting security controls so that people are able to get their work done is a way that we can get what we want (secure users) with less conflict. *User feedback matters.*

NOTE Examples of insecure user workarounds for inconvenient and/or unusable security policies: users writing passwords on sticky notes because the security team forces them to rotate their passwords every 90 days; applications preventing screenshots, resulting in users taking pictures with their personal devices to share information with their colleagues; scanning code with an untuned static application security testing (SAST) tool that provides more false positives than true ones results in it being disabled in the pipeline, and the code is then pushed with no security testing. The list is endless.

Fail Closed/Safe, Then Roll Back

Whenever there is a problem during a transaction, it is important that we roll it back and do it again. If we were opening something (such as a database connection), we should close it. If we were granting access, we should revoke it. We do this to avoid race conditions, timing attacks, and other potential vulnerabilities. Going back to the start and doing it all again requires very little effort or time with a modern computer. But allowing an error or failure to continue can allow our system to fall into an unknown state. And that is where vulnerabilities often lie in wait: when our system's next action is not predictable. Thus, it is always safer to fail closed, roll back, and start again.

Compliance, Laws, and Regulations

Compliance: the act or process of complying to a desire, demand, proposal, or regimen or to coercion.

— Merriam-Webster

There aren't very many laws related to cybersecurity outside the United States and the European Union, and as far as I could find on the internet, none of them concentrate solely on the security of software. There are, however, several that cover data security from the United States and EU, which are very closely related to software: the Federal Information Security Modernization Act of 2014 (FISMA), the Health Insurance Portability and Accountability Act of 1996 (HIPAA), the Notice of Security Breach Act of 2003 (only applies to California), the EU Cybersecurity Act, the Directive on Security of Network and Information Systems (NIS Directive), the Digital Operational Resilience Act (DORA), and the General Data Protection Regulation of 2018 (GDPR).⁷

That said, there are several self-appointed regulating bodies in the field of information security that offer compliance and standards we can follow. Although many in the industry will agree that security and compliance are not the same thing, creating a requirement for users to follow best security practices can be quite helpful to propel organizations to develop and maintain a more robust security program.

What follows is a non-exhaustive list of compliance and standards in the field of information security. You do not need to memorize them all, but being aware that they exist and if your organization needs to comply with any of them will help you do your job better:

- Payment Card Industry Data Security Standard (PCI DSS):⁸ For companies that handle credit cards and other forms of payment online.
- NIST special publications: For any information systems, there are several different standards available from NIST.
- New York Department of Financial Services (NY DFS) Cybersecurity Regulation: For financial organizations from the United States.
- The Digital Operational Resilience Act (DORA) is a European Union regulation that aims to strengthen the IT security of financial entities, including banks, insurance companies, and investment firms. This regulation is set to apply from January 17, 2025. DORA focuses on ensuring that the financial sector in Europe can maintain resilience in the face of significant operational disruptions.
- International Organization for Standardization (ISO) standards: A system built to help organizations manage their entire information security program.
- General Data Protection Regulation (GDPR): A privacy-focused regulation that aims to protect consumers.
- Health Insurance Portability and Accountability Act (HIPAA)/Health Information Technology for Economic and Clinical Health Act (HITECH)—The protection of personal health data of US citizens.

- Federal Information Security Management Act (FISMA): Another regulation from the United States to help protect against data breaches for governmental organizations.⁹
- California Consumer Privacy Act (CCPA): Protecting the privacy rights of California residents.

Security Frameworks

There are several organizations that have created security frameworks in an effort to make system processes easier to define, implement, and maintain. What follows is a non-exhaustive list of some frameworks you may want to learn about on your journey into secure coding and secure applications.

An unsurprising first choice are three frameworks from OWASP (the Open Web Application Security Project): the Security Knowledge Framework, DevSecOps Maturity Model (DSOMM), and OWASP SAMM (Software Assurance Maturity Model).

The Security Knowledge Framework¹⁰ is a free and open-source software that uses OWASP's Application Security Verification Standard (ASVS) as its base for helping developers write secure applications. It gives secure coding examples in several languages so that programmers can look up the most secure way to do whatever they are trying to accomplish.

OWASP DSOMM is a framework that illustrates various security measures that are easily applicable when using the DevOps software development methodology and how to prioritize them.

OWASP SAMM¹¹ is a framework for helping you evaluate the maturity of your secure system development life cycle (S-SDLC). OWASP SAMM is agnostic of any technology or programming language; it's just there to help you improve your SDLC and other processes to ensure that your software is secure at the end.

OWASP offers several other frameworks, including a web app testing framework. It also has endless documentation about application security, shared freely on its website, <https://owasp.org>. Its best-known resource is the OWASP Top Ten Security Risks to Web Applications (affectionately known as “the Top Ten”), which can be found here: <https://owasp.org/www-project-top-ten/>.

NOTE When giving security advice to software developers, it is extremely important to be timely, specific, and concise. Back in the day, I worked on a security team with over 20 people, who mostly performed risk assessments that were made up of checklists. When developers asked for advice on their applications, the security team provided answers that absolutely baffled me. A software architect asked if he could be given the list they use at the end of the SDLC to validate the security of his application so that he could build it in from the start. He was told “that would be cheating,” and

his request was rejected; apparently the security criteria were a secret. One developer asked for details of how to securely manage his application's session ID and was sent a link to the NIST website, with instructions to read the entire thing (easily over 1,000 pages) so he could learn how to build secure apps. I tried to explain to the security team that this was, effectively, like sending a picture of them giving the middle finger to the developers who sent these questions. It was both *unhelpful and insulting*. No one agreed with me. I ended up leaving that workplace rather quickly; I just didn't fit in.

Up next is NIST, which releases many frameworks along with other extremely helpful publications; NIST is an amazing online resource. NIST special publication (SP) 800-53¹² is an application security framework that describes the recommended risk management practices for software. This publication focuses mostly on how to test software and tools that are recommended for such testing. Most recent updates include newer tooling such as Interactive Application Security Testing (IAST) and Run-time Application Self Protection (RASP) tools.

Also from NIST is the Cybersecurity Framework (CSF), which states on the title page that it is for improving critical infrastructure, but in reality it can apply to most IT systems. This framework focuses on managing and evaluating risk rather than specifics on secure coding or design.

Another group you should be aware of is the Center for Internet Security (CIS), which created a list of Critical Security Controls.¹³ Previously known as the "SANs Top 20," this list of controls is the biggest bang for your (security) buck at a large organization. It includes controls for securing all of an organization, not just IT or software development. This group also releases regular "CIS Benchmarks," which are vendor-neutral security configuration guides, as well as pre-hardened infrastructure images. CIS focuses on infrastructure over custom applications.

The next two are closer to compliance, but technically they are independent bodies. That said, they are usually implemented to prove security maturity to those outside of their organizations, such as customers, investors, and business partners.

The International Organization for Standardization (ISO), ISO 27001, and ISO 27002 certifications are considered the international standard for validating a cybersecurity program.¹⁴ It might not be a framework per se, but it guides you through the steps to get your house in order with regard to risk management.

Service Organization Control (SOC) Type 2, also known as SOC2, is a list of 60 controls for securely managing client data.¹⁵ The controls for this framework are intense, and it's a lot of work to implement, but when an organization can state that it has "SOC 2 Compliance," it is a huge selling point for any product/company.

Although there are certainly several more cybersecurity-related frameworks that exist, none of them are related to the security of software. This leaves us

with OWASP and NIST as leaders in providing software-focused security frameworks at the time of this writing.

Learning from Mistakes and Sharing Those Lessons

Although this topic is not strictly about security, many cybersecurity issues occur due to a lack of knowledge, information, or training. If every security team made a point of sharing information widely within our organizations when a bug was fixed, a security flaw was discovered, or a postmortem was performed after a security incident, we would be less likely, as an industry, to fail so often. With cybersecurity issues, such as data breaches and other failures, happening so often, we must work hard to avoid repeating our mistakes.

Backward Compatibility (and Potential Risks It Introduces)

As someone who prefers to hold on to my personal devices for as long as humanly possible, I value backward compatibility and the cost savings it provides to the end user and my personal bank account. That said, it is not without risk. Software ages very badly; the longer something is in the public eye, the more potentially malicious attention it may have received. Vulnerabilities are sometimes discovered immediately, but often it takes time, and the more time provided, the more likely vulnerabilities will be found. As software, hardware, and their dependencies age, some of them stop being supported, which also has negative security implications (no more testing, no more security updates, etc.). When you are deciding how long to provide backward compatibility for a product you have created, please take security into account and whether you can continue to ensure that it will be safe for the end user.

Threat Modeling

Threat modeling is a process for identifying threats to a system you are charged with creating or maintaining. You identify the potential threats and then analyze whether you should mitigate them or not (some threats are minor or highly unlikely and do not require fixing to maintain your desired security posture). When you perform code review and security testing, you should search for your threat models and try to prove that your system is adequately defended against them.

Threat modeling can be incredibly formal, with a lot of documentation, processes, and time involved. It can also be informal, such as a brainstorming

session with some note-taking. Whichever way you choose, this process is almost always eye-opening, illuminating several potential issues you had not previously planned for. We will go into further detail on this topic in the third part of this book.

PLANNING FOR PRIVACY

Years ago, when Alice was much younger, she dreamed of being a famous movie star. She auditioned and got small parts in a few independent films and even had a small following of fans. It didn't last long, but she enjoyed it while she could. During this brief stint, an older, more experienced actress took her aside and explained that now that she was "in the public eye," her threat model had changed. "What's a threat model?" Alice asked. The older woman told her that she would receive mostly good attention, but that with any level of fame it also attracts negative attention, and thus it was very important to protect her personal information, such as where she lived, her date of birth, her social insurance number, and anything else someone might use to find her or impersonate her. Alice took this advice seriously, and as she got older, she was very conscious of what images and information she shared online. Having fans was great, but she didn't want to ever be surprised by one of them showing up at her front door unannounced!

The Difficulty of Patching

When we apply updates to our phones or software on our computers, it is called *patching*. When we update (patch) our operating system, sometimes it can cause software to have issues or require that older software be updated or it will no longer work. When there is an issue in a dependency for software, if it has its own dependencies, it can mean updating several at a time, which can be complex, difficult, and time-consuming. Now imagine updating a programming language framework on your custom software. Sometimes this means you need to swap out dependencies or not call certain functions anymore; and once in a while (usually when you are several versions behind), it can require a complete rewrite of your application! After that work has been completed, extensive testing is required to ensure that the application continues to work as intended and that you have not created new bugs (security or otherwise) in the process. Please keep this in mind when someone says, "Just patch it"; it is usually much more work than meets the eye.

When working with companies on their application security programs, the complaint I hear most often is, "The developers won't fix the vulnerabilities we report." When we dig a little deeper, the reason is often one of the following

two issues: the software developers have an inappropriately high workload and don't have the time to fix the issues, or the release process is painful, time-consuming, and ineffective. More than once, I have told clients, "Your main threat to your software security is your inability to release software fixes in a reasonable amount of time. We need to fix this process as soon as possible." Anything you can do to streamline or automate this process will help your organization respond faster to security issues and is a good investment of time and money.

PRIVACY THAT DOESN'T WORK

Bob had always loved his MacBook and the Apple MacOS operating system, but when the Catalina MacOS Update came out in 2019, it provided all sorts of new privacy "features" that broke almost everything! He was supposed to give a presentation at work the next day, and his presentation software couldn't access his folders without him messing around for 10 minutes; he was very embarrassed. He tried to make a video that evening on his personal machine, and Catalina struck again! He had to get completely new video recording software; Open Broadcaster Software (OBS) was completely nonfunctional. It took him weeks to get his personal machine back to normal and have everything working the way he liked it. Bob has a lot of sympathy for people who apply patches for a living!

Retesting Fixes for New Security Bugs

Whenever we fix a bug, we are changing and/or writing code. When we write new code or change existing code, it is very possible that we can create a new bug. And some of those bugs are security bugs. It's never on purpose, but it happens to everyone at some point. The point here isn't to shame anyone for making errors; it's what human beings do. The point is that we must always retest our code after we make changes, to avoid this situation whenever possible. Ideally, we do this retesting using automation. With the advent of DevOps and continuous integration and continuous delivery/deployment (CI/CD) pipelines, doing this is easier than ever before. On top of this, there are now hundreds of security tools that can be automated to retest your application in very little time, including free and open-source options. Keep reading this book, and we will go over how you can test your own apps without the need to be a security expert.

NOTE **KISS: Keep It Simple, Silly Complexity is the enemy of security.** Systems work better when they are kept simple. This applies to everything, including the applications you develop. The more complex the code is, the easier it is to make errors, and the more difficult it is to maintain. Reducing complexity will help you create more secure software.

Chapter Exercises

You can find the answers to these questions in Appendix B, “Answer Keys.”

1. Why is it important to retest our code after we fix a bug?
2. Why is it important that we (try to) keep our programming frameworks up to date for custom software that we build?
3. Which security framework seems the most useful to you, and why?
4. What is your personal threat model? (Note: There are no wrong answers here. The point is for you to identify potential threats in your life and see if you can mitigate them.)
5. Give one example of “assumed trust.” It does not have to be IT related.
6. Think of one new security thing you have learned that you think would benefit your team at work, at school, or in your personal life. Share that information with the appropriate people. It can be something from this book or anything else you think might help someone be more secure.
7. Give one example of least privilege and how you would apply it. Explain why, and the potential consequences if least privilege was not applied to this situation or system.
8. Give an example of a potential secure default or “paved road” that you would like to implement.
9. Give one example of security through obscurity.
10. Give an example of a situation where you would “fail safe,” “fail closed,” or “roll back a transaction” and how it could provide protection to the users of that system.

Beginning

Secure coding is a bit like muscle memory: the more often you do it, the easier it becomes. You make a habit of validating all the input to your app. You always use parameterized queries. You meet with someone from the security team to ask them to help you classify and label your data, until you know how to do it by yourself.

In this part of the book, beginning with this chapter, we are going to cover secure coding advice that applies to the majority of programming languages and frameworks. The information in this chapter is the start of your secure coding journey.

Follow a Secure System Development Life Cycle

If I could give only one piece of advice to a company, team, or person, it would be to follow a *secure* system development life cycle. What do I mean by that? I mean adding security activities as part of the processes you follow to develop software. This could mean starting off your project with a list of requirements that includes security requirements and then, during the design phase, doing a threat-modeling activity with your AppSec person and product manager. There are all sorts of activities that can be added as part of your regular processes to help ensure that you have more secure software when you finally release your app. In Part III of this book, we will cover how to do this in depth.

BOB'S FIRST SECURE SYSTEM DEVELOPMENT LIFE CYCLE (SDLC)

Bob remembers the first time a security team wanted to “add stuff” to his project. A woman from security showed up at a project kick-off meeting. She said, “Hi, I’m Alesha, and I will be your security resource for this entire project. I will help you arrange or do all of the security requirements we have for this project. I will review all of your documentation and offer assistance during the whole project. If you need help or have questions, I’m your person.” Bob was so surprised! Usually, the security team kept their distance, and this person was new. She kept her word, though! During the entire project, she kept showing up, checking in, and helping. Sometimes Bob would ask questions, and she would admit it if she didn’t know the answer and then arrange a time for them to brainstorm and research the problem together. She never gave up on him and his team. It was a different and *better* experience for Bob. Later, Bob learned that this is called “The Partnership Model,” and it became his favorite way to work with the security team.

Use a Modern Framework and All Available Security Features Within

The next piece of advice is to use a proven, modern, industry-standard programming framework whenever possible. Sometimes we don’t have a choice, and we have to use whatever everyone else in our org is using. But if you have a choice, pick your framework carefully, and if possible, choose one with a better reputation when it comes to security.

When using a programming framework, try to always use the security features built into it instead of writing your own. The reason for this is that programming frameworks are tested extensively—not only by the teams that create them but also by the programmers who use them all day long. With such extensive testing and usage, they are more likely to be secure than code written by one single programmer off the side of their desk.

SCOPE CREEP FOR ALICE

In college, Alice took a programming course, and she had to do a big project with a team of classmates to pass the course. One of her teammates decided he would “write his own XML parser.” He said it would be easier than using one that already existed. It turned out *not* to be easier. In fact, his parser took so long that he never finished it, and the rest of the team had to find one and implement it at the last minute as their deadline loomed. The whole 4 months, he kept telling them he was “almost done” and that *his code* was “way better” than the XML parser the team had chosen. Alice didn’t understand: why would he do this?

As she continued her career toward becoming the executive she is today, she saw this over and over: programmers thinking they could do better than an entire team

of dedicated professionals who created whatever product the developer(s) wanted to build instead of buy. They would reinvent the wheel all the time, and it cost a fortune! The maintenance of homemade apps was never considered as part of the cost, and it caused a lot of arguments with upper management.

Input Validation

Now let's get into *coding* advice. Whenever we accept input into our application, we should check to see if it's what we are expecting and that it's safe to use; this is called *input validation*. Input can be from anywhere: the user entering text into a search field, data returned from an API or database, an uploaded file, someone changing something in a hidden field, or manually changing a URL parameter. If there's a way for a user to change it, you should assume it might be changed. All of these inputs could be dangerous to your application, whether intentional or not. It could be a user who leaned on their keyboard by accident or a malicious actor with bad intentions. Because we cannot read minds or predict the future, the easier way to defend against such situations is to always make sure the data is safe to use before we do anything else in our software.

To do this, we want to start by validating that the input is *what we expect*. We should do this on a trusted system (the server, rather than in the browser's JavaScript), and we should do this using an allow list (values/patterns/formats we are expecting) rather than a block list (anything we are afraid will be used to harm our app). Note that we can *also* validate the input in the front-end JavaScript, but not *in place of* validating on the server, only as an additional check.

Let's break this down a bit because if you get this right, your apps will be significantly more secure. Proper and thorough input validation can eliminate a very large number of vulnerabilities in any piece of software.

First, what are we expecting to receive? Let's say we are expecting a date, and the format is MM/DD/YYYY.

In Visual Basic for Applications, we can check if something is a date by using `IsDate`.¹⁶

```
Dim MyVar, MyCheck

... Code that gets a date from the user ...

MyCheck = IsDate(MyVar)
' returns true if it is indeed a date, false if not
```

Then we will likely want to check if it's in a range that makes sense. Assuming that we are asking for a person's date of birth, we expect to receive a date that is before today and no more than 150 years ago.

TIP Cross-site scripting, often called XSS, is a very dangerous and very common vulnerability in web applications. It happens when a web browser is tricked into interpreting the attacker's code as if it were part of the application, after which the browser attacks the end user on their own computer. The constraints of JavaScript are the only limits on the awful things that can result from this vulnerability, such as installing a keylogger on the end user's computer, turning on their webcam and recording them, or installing malware. We will dive into this vulnerability and many others in Chapter 8 of this book.

Moving over to ASP .NET Core and C#, our code¹⁷ could look like this:

```
DateTime MyDate = new DateTime();
// assigns default value 01/01/0001 00:00:00

MyDate = ... code to get date from user... ;

If (MyDate.TryParse(text, out dateTime)) //is a date19
{
    if (MyDate >= DateTime.Today) //date is today or later
    {
        Console.WriteLine("Invalid date of birth. Please try again.");
    }
    else if (MyDate <= DateTime.Now.AddYears(-150);)
    { // Minus 150 years from today18
        Console.WriteLine("Invalid date of birth. Please try again.");
    }
    else
    {
        // code for the rest of your app, your date is good
    }
}
Else
{
    Console.WriteLine("Invalid date of birth. Please try again.")
}
```

TIP We always validate and encode, but we only sanitize and escape when we need to accept potentially dangerous characters.

We've checked if it's a date, and then if it's earlier than today but later than 150 years ago. In doing this, we have validated that what we have received is *what we were expecting*, and therefore it's safe for us to use.

In Chapters 6 and 7 of this book, we will go over popular programming languages and frameworks, with more examples and things to watch out for.

Usually, we can just validate the data and either reject it (if it fails validation) or accept it. Unfortunately, sometimes life is more complicated than this, and validation won't work.

When you *must* accept special characters, things can get a little dicey. Some of the other defenses you can use include sanitizing, escaping, and then encoding (in that order). Note that you should always validate and encode for web apps.

TIP Even though this book is written in English, the entire world does not operate solely in this one language. Be careful not to block other alphabets that your users will need when you are choosing which characters to accept.

- **Sanitizing** input means removing or replacing any problematic characters. An example of problematic characters for JavaScript would be `<` or `>`. The reason they can be problematic is that they are used to start and end a `<script>` tag, which then could be used to attempt a cross-site scripting (XSS) attack. You could easily just remove them, or, if you needed to know they had been there, you could make your app replace all the open and close bracket characters with something else instead (perhaps a letter or a not-dangerous special character such as a tilde symbol).
- **Escaping** means adding a `\` (backslash) in front of special characters before saving. This removes any special powers the following character has. If the information is later run through an interpreter, it would not give special powers to the escaped character(s).
- **Encoding** is similar to escaping in that it takes all the special powers away, but it does it to everything that you encode, not just one character. But there's more to it than just that. Stay tuned!

Ideally, you would validate as much of your input as possible and then only sanitize or escape it if you must accept special or potentially dangerous characters. You should always encode information being put onto the screen for web applications.

Let's look at another example. Imagine you are programming in PHP, and you want to validate some input from a search field. You can grab the value to search for from your user, but how do you make sure it's what you are expecting?

The following code is an edited version from w3Schools:²⁰

```
<?php
// define variables and set to empty value
$name = "";

if ($_SERVER["REQUEST_METHOD"] == "POST") {
    $name = validate_input($_POST["name"]);
    //get "name" from form, send for validation
}

function validate_input($data) {
    $data = trim($data); //trim off spaces and tabs
```

```
$data = stripslashes($data);      //remove backslashes
$data = htmlspecialchars($data); //convert special characters
return $data;                      //return safe-to-use data
}
?>
```

A few thoughts on input. Any data fed into your app or API could be an attack vector. Quite often, when mistakes are made for input validation, it's because a software developer didn't think of it as input, and they skipped the validation step.

For instance, you use a stored procedure to get some data from your app's database: that can't be malicious, right? Wrong. Unfortunately, if someone didn't do proper input validation when putting the user data into the database, someone else could have stored an XSS attack in there, waiting for you to retrieve it and be fooled.

We also want to ensure that the data we retrieve is the correct type (test it; don't just assume it is the type you have been told it is) as well as that the range of data we are receiving is logical and possible; make sure it's what are you expecting and the values are reasonable. We wouldn't want to have a person who is 500 years old or -5 years old.

Another place developers tend to forget to put appropriate security controls (such as input validation, authentication, authorization, etc.) is an API that calls another API or serverless app. Often the developer assumes the data coming from their own API or app is safe to use. But what if that app was compromised? What if the API was poorly programmed? What if someone has replaced your app with their own app (less likely, but still possible)? There are many situations where we assume trust, which can lead to software vulnerabilities. With this in mind, always validate your data before you use it, no matter where you get it from.

Output Encoding

As programmers, when we output something to the screen, we choose what format it is displayed in. This is usually based on visual appeal / user experience (UX) and tech stack (is this HTML? JavaScript? A Windows app? A text-only backend system?). However, when dealing with web applications that are run from within a browser, we have more to worry about than just looks.

Browsers execute (run) your web application, just like Windows runs local Windows apps, Linux runs apps made for Linux, iOS runs Apple iPhone apps, Android runs apps for Android, and MacOS runs Mac apps. The difference between web apps and most other kinds of software is that 1) they are generally viewed over the Internet or a network of some kind (as opposed to natively on whatever operating system you are running), 2) they are interpreted on the spot, rather than saved into an executable that is then installed onto your operating

system and run from/by your operating system, and 3) they need a browser to run (it has the interpreter!). A web app can be run by many different types of browsers, and each of them has slightly different functionality.

When browsers run your web apps for you, they are interpreting the apps on the fly. The browser takes all of your HTML (hypertext markup language), CSS (cascading style sheets), and JavaScript (not to be confused with Java), interprets it, and presents the app for you to use. It interprets your code in the browser as your users perform various tasks. It will interpret it in different contexts based on your context within the app. If you are in the HTML part, the browser will use the HTML part of the interpreter, and JavaScript if you're in the `<script>` part. All of your CSS is also interpreted by the browser.

Sometimes, when browsers are interpreting your code, they get confused and mistake user input for code instead of data. Data is handled, saved, changed, and used to make decisions within applications all the time, but it should never be executed or interpreted as though it is part of your application code. If user input is mistaken for code and then interpreted by the browser by accident, this is called cross-site scripting (XSS).

You may have heard of XSS before, and if you have, that's great! This means you are already aware of at least one of the common risks to web apps. If you have not, you should be aware that it is generally considered a danger to web applications (all of them); it happens far more often than any security professional feels comfortable with, and when we escape our data before putting it on the screen, we can avoid this problem most of the time.

To avoid XSS, it is best to perform input validation, output encoding, use the `Content-Security-Policy` header, and, ideally, also perform code review and security testing to verify that you implemented the previous precautions correctly. That said, if you only have the time, money, and/or enough influence for one defense, output encoding is the big winner when trying to avoid XSS.

When performing output encoding, it is important that you encode for the context you are in within your code. If it's inside an HTML tag, that's your context (meaning you use HTML encoding). If it's in CSS, that is your context. Sometimes you are in a tag within another tag, and it is possible that you may need to encode twice (I realize that this is not the most fun code you may ever write, but trust me, preventing XSS will help you avoid a potentially very bad day in your future).

NOTE Always use the encoding features in your framework, if available, rather than writing your own.

Examples of Output Encoding

Most of my XSS encoding examples are from the Open Worldwide Application Security Project (OWASP) XSS Prevention Cheat Sheet.²¹ The Cheat Sheets

Project is the first resource that I always start with when giving examples or advice. I highly recommend the entire Cheat Sheet series and commend the people who maintain this wealth of excellent advice. You can see more at https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

HTML Context

Let's say you want to insert a variable between two HTML tags. This is the HTML context, meaning you need to encode it for HTML (which means changing it into character entities, which have no special powers). You could write some JavaScript to manually change each < character into `<`, which is the HTML-encoded equivalent. You could write an entire script to change every character, one at a time. But most software developers are not writing their HTML from scratch; they usually use .NET, PHP, or something similar. So let's look at how you can generate safe HTML and JavaScript for this situation.

TIP If you are putting a variable into HTML, CSS, or JavaScript tags, you must validate it, potentially sanitize or escape it, and then encode it. This is a high-risk coding situation; proceed with caution.

.NET

In the .NET and .NET Core frameworks, there's a handy function for this called `HtmlEncode`, which does exactly what it sounds like it does:²²

```
public static string HtmlEncode( string s )
```

PHP

In PHP, you need to manually encode your output, like the following HTML encoding example:

```
string htmlentities ( string $string [, int $flags = ENT_COMPAT |  
ENT_HTML401 [, string $encoding = ini_get("default_charset") [, bool  
$double_encode = true ]]] )
```

JavaScript Context

When JavaScript is encoded, it turns into Unicode. Think < turning into `\u003c`.²³

.NET

Luckily, the .NET framework created the `HttpUtility.JavaScriptStringEncode` function to do all of the encoding for us. Thanks, MSFT!

PHP

Unfortunately, PHP doesn't offer an out-of-the-box solution for this; you must write your own code. The easiest way to do a great job of encoding is to use frameworks that do it for you. A very large part of the software developer population use JavaScript frameworks such as Vue.js, Angular, and React, which automatically take care of this for you. They automatically output encode, escape, and perform lots of other security features without you having to think about it. Whenever possible, avoid writing your code from scratch and instead use a modern programming framework.

Parameterized Queries and ORMs

When I first became a software developer, I never really thought of a database as being an interpreter for my code. When I wrote my first few glorious lines of SQL, it never occurred to me that it could be used for evil, not just to fetch records to show my users. As a programmer, my focus was on solving problems for my clients by building software that served them and their business needs. Way back then, I did not fear having my queries used against me, my app, or the users of my apps. I was so wrong!

When I was working for the Canadian federal government, we had an ethical hacker (also known as a penetration tester) give a presentation for my dev team. He showed us a login screen that looked just like one of our web apps. He said, "I'm going to log into this app, with no password, in one minute. And it's only going to take me that long because I'm talking." I thought, "Sure, buddy. . .," but then he did it!

Let me tell you how he did it. First, he said, "What do you think is behind this screen? A query that looks something like this:

```
CMD = "SELECT * FROM TABLE USERS WHERE USERNAME LIKE '%" + USERNAMEVAR + "%'";
```

"Now imagine if I entered your username: 'SheHacksPurple'. Once the SQL server gets hold of it, it will look like this:

```
SELECT * FROM TABLE USERS WHERE USERNAME LIKE '%SheHacksPurple%'
```

"It will return all the usernames that contain 'SheHacksPurple'. But what if I put *code* into the *username* variable instead of a name? What if I put something that always returns *true*? Something like this:

```
USERNAME = " SHEHACKSPURPLE ' OR 1=1 - -";
```

"Then the code sent to the SQL server will look like this:

```
SELECT * FROM TABLE USERS WHERE USERNAME LIKE '%SheHacksPurple' OR 1=1 - - %'
```

“This command will look for records where someone has a username of ‘SheHacksPurple’ or . . . every single other record in the database! Because `1=1` is always true. What this means is that if your code looks to see if the username exists in the database, it will return `true` every time, whether that user exists or not.”

Once he showed us this, he explained that we could do the same for the password field, and then we were in. Then he told us this type of attack was called SQL injection (SQLi) and that there were several other types of injection, such as command injection, LDAP injection, and even NOSQL injection (for non-SQL query languages). He said it was the number-one worst vulnerability for web apps and that it is on various sites all over the Internet right now! I was flabbergasted. This lesson was my first step toward deciding on a career change from software developer to application security professional.

NOTE Adding the two dashes at the end (`--`) means we have ended the SQL statement early, and we can disregard everything after the two dashes. This is used a lot in this type of attack (SQLi) to simplify the attacker code. Why bother ensuring that your attack code is syntactically correct for the entire query when you can just cut it off after the first (vulnerable) parameter?

We could have prevented this attack from happening by using a combination of input validation (first layer of defense) and (second layer) parameterized queries (as opposed to inline SQL or whatever other database query language you are using, including NOSQL).

You might be thinking, “That’s not that bad; I can write my login code very carefully. I will be fine.” Unfortunately, injection is a very dangerous vulnerability that should not be underestimated. SQL injection alone can allow an attacker to steal all your data, update or delete your data, or (in combination with other vulnerabilities) take over your SQL server so the attacker is within your network, behind your firewall, controlling your server. Injection vulnerabilities allow for extremely dangerous attacks against applications, operating systems, and networks.

BOB’S FIRST FORAY WITH STORED PROCEDURES

Bob remembers when the security team told his dev team they couldn’t use “inline SQL” anymore and had to use stored procedures from now on. Someone had to explain to him that *inline SQL* meant him using his own code (including user input) to create an SQL command and then sending it to the database to be executed on the server. The security team wanted him to make a parameterized query instead. It meant he would have to call a function inside an SQL database from his app and then send the data to that function inside a parameter. Bob didn’t understand the difference: *Who cares where the query is written?* The security team explained that

when you pass the data as a parameter for a stored procedure, the server removes any special powers it might have, and that ensures that it is only treated as data, making injection attacks impossible. Although Bob didn't really understand how the server managed to do that on the backend, it worked! After switching to only stored procedures, his team passed all their security tests, so he was satisfied with the outcome.

Authentication and Identity

Authentication is how a computer or system verifies that a user is who they say they are. They want to find out if you are the real, authentic you and not an imposter. When someone walks up to your car door or rings your doorbell at home, generally you look at them, figure out who they are, and decide if you want to let them in. Sometimes people with vision impairment will use a person's voice to recognize them, or they will check the person's identification to find out if they are someone they should trust. Most people do not let strangers into their homes, cars, and offices. But computers don't come with senses (like our bodies do) so they can recognize someone; they need another system, mechanism, or device to do this. The digital system they use to do this is called authentication.

AUTHENTICATION AND AUTHORIZATION: TWO EXTREMELY IMPORTANT SECURITY CONTROLS

Quite often, you can hear security professionals referring to AuthN and AuthZ, assuming everyone knows what they are talking about and what those weird-sounding short forms mean. Wonder no longer, dear reader!

- AuthN stands for authentication. It's the computer asking, "Who are you?" and deciding if you are one of its recognized users.
- AuthZ stands for authorization. This is the computer's decision-making process to ask, "Should you be here?" and then grant access or not.

Generally, computer systems authenticate and *then* authorize a user.

NOTE AAA stands for authentication, authorization, and accounting (or auditing). Sometimes referred to as IAAA (with the "I" standing for Identity), the main addition here is auditing of the authentication and authorization: *the act of verifying that the access is being used correctly.*

Identity is when a person (user) or a system claims an identity on a network or other digital system to distinguish themselves from other users and systems. As you go about doing things on a network or other digital system, if it

needs to check who you are, it checks your identity (just like they check your identification when you go to the airport, a doctor's office, etc.). Everything on a network can have an identity that is kept track of with an identity management system. Even a web application or API will have an identity.

The most-used identity system is Windows Active Directory (affectionately known as AD). Other systems on the market include Okta, Microsoft Azure Active Directory, Auth0, and SolarWinds.

NOTE SSO stands for single sign-on. When using SSO, a user signs in when they start using the system, and then they are granted access and use of the other systems that are part of the SSO system. I first had a chance to be part of a team to create one of these when working at the Treasury Board Secretariat as part of the Canadian federal government. We had several applications that all public servants could use, and we didn't want them to have to sign in 20 times a day. We allowed all of them to sign in once, using SSO, and we had the applications pass a token, which included the user's identity, in the background. The user didn't have to do anything to make this happen, and that's why SSO is a great example of *usable security*.

Now that we know what authentication and identity are, how do we secure them? Ideally, you buy a system instead of building your own. These systems are incredibly complex, and if you create your own, there is a very good chance you could make a mistake. The systems on the market have been tested very thoroughly, and sometimes people still find bugs. The other security-related reason you often buy these systems is that this is the first and more important defense of your system, meaning it will be tested (by bad actors) the most often. A nonsecurity-related reason to buy such a system is that creating a good system for this task would be more expensive than buying one. The amount of time to create and also maintain such a system would not be worth it for most situations.

Examples of SSO provider platforms include Facebook, Google, Microsoft, LinkedIn, and X (formerly known as Twitter). Examples of SSO systems you can buy and implement yourself include Duo, Indaptive, Okta, NetIQ, PingOne, and OneLogin.

For details on how to write your own custom system, see *Alice and Bob Learn Application Security* (Chapter 4, pages 94–98) and the OWASP Authentication and Identity Cheat Sheet (https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html).

Authorization and Access Control

Authorization is a computer deciding which digital access or permissions you are allowed. Access control is the technical control that grants or restricts access. We are going to go over what this means, but your first takeaway is that (whenever possible) you should buy a system that does this for you rather than

building your own. If there are features for implementing a system that is part of your framework, use that as a second choice rather than building one from scratch, which should always be your last choice.

NOTE Buy, borrow, build, in that order. Always buy a system that does the thing you need and only then, if one is not available that meets all your requirements, borrow one. Borrowing can mean reusing code from another team, another organization, or using the features in your framework. As a last resort, we build from the ground up, writing each part of the code ourselves. We prioritize in this order to save time and money but also because prebuilt systems and framework features generally have more security attention and thus are a more secure option. It's a win-win all round.

Look at Figure 2.1: Is Alice allowed to see this folder? Can she edit what is inside? There are *models* for access, and then there are *implementations* (logic access control methods) for access.²⁴

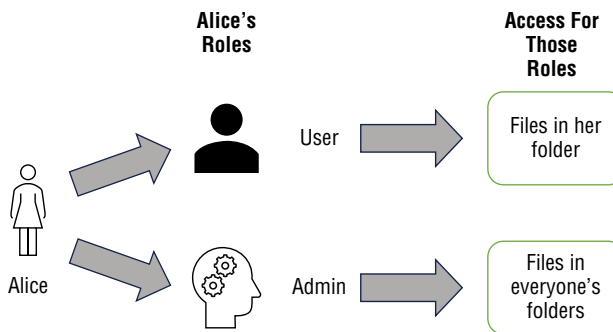


Figure 2.1: Access control for Alice

Access Control Models

Although there are many different models, the following four are the best-known.

With **mandatory access control (MAC)**, labels of classification (such as secret, top secret, and unclassified) are assigned to people, resources, systems, and other data, and then only those who have the matching level of authorization (or above) are allowed access (for instance, if you have top-secret clearance, you are allowed to view top-secret documents that apply to the projects you are assigned to and the work that you do). This model is often used in governments and especially in military situations.

Role-based access control (RBAC) provides access based on the role a person plays within their organization. For example, if you are a database administrator (DBA), you will be granted access to some or all of the database servers. And if you are not a DBA, you will not have access to any database servers.

Discretionary access control (DAC) allows the creator of a system, document, or program to have complete control over it and to grant others access. This is commonly used in online office tools such as Office 365 and G-Suite; you create a document and then manually decide whom to give access to what you created. DAC is used almost everywhere. It is the most common form of access control in consumer systems (Windows and Linux, for example).

Rule-based access control (RBAC or RB-RBAC) dynamically grants access based on criteria set by the system administrator. This could mean allowing daytime employees access to their systems only from 8:00 a.m. to 6:00 p.m. each day, only allowing devices to connect from a specific IP range, or removing all access immediately when an employee's status switches to "terminated." Firewalls are a great example of this type of access control.

Logical Access Control Methods (Implementation)

Access control lists (ACLs) are lists of identities on your systems that are allowed to access objects on your network or within your systems. Look at Figure 2.2: "Can Tanya edit this document? No. Can she make a comment? Yes. Great; she will add comments now."

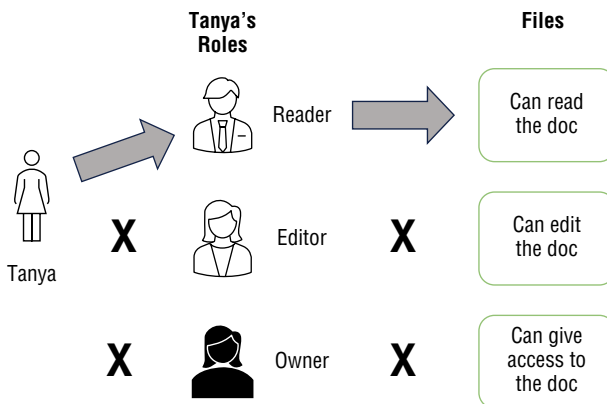


Figure 2.2: Access control for Tanya

Group policies allow people within various groups (electronic lists) to perform certain actions (this is a part of Windows OS). Group policies can be applied to users, systems, or groups.

Biometrics, PINs (personal identifiers), passwords, and tokens are other logic controls that can be used to grant or restrict access.

Session Management

When a user moves throughout your application, clicking buttons, links, and other parts of your app, this behavior is tracked via a session, the technical

details of which are generally stored in your cookies. We design the app to do this so it can keep track of various actions our user has taken, remembering what the user has done within our system. If they have selected a very nice pair of shoes and put them in their shopping cart, when they go to the next page, the user wants the app to remember the shoes in the color and size they selected. When they (hopefully) click the “check out” button, the user wants the app to remember all the items that they put into their cart. This entire time, we also want the app to be aware that it is the same user, not another IP address/user/imposter. This is called a *session*.

Sessions are not an original feature of the Internet. When the Internet was designed, none of its creators had any idea it would turn into what it is today. Because sessions are not one of the original features, it is not built into the protocol (TCP/IP) or any of the other building blocks of the Internet.

If there is a session management feature available in your framework, use it. If not, you want to pass your session ID and other information back and forth (from client/browser to server) in a secure cookie. There are extensive instructions on how to do this in *Alice and Bob Learn Application Security* in Chapter 4, under “Session Management” and also in the OWASP Session Management Cheat Sheet (https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html).

Secret Management

Secrets are digital authentication credentials (how the computer proves who it is). Secrets are used by computer systems to access other systems, data, networks, etc. Secrets are not used by human beings; they are used by computers to authenticate to other computers. The best-known type of secret is a password, but there are also keys, API secrets, tokens, hashes, certificates, and more. When security professionals say “secret management,” we mean, how are these secrets handled? Stored? When one has expired, how do you rotate it for a new one? A secret management tool performs the task of managing secrets for you.

Back when I was a programmer, I had never learned anything about this subject in school or at work. I stored all my connection strings (the code that you use to connect to a database) in my code. My whole team did. We would have a connection string for production, one for the development environment, and usually a few other environments as well, all saved right into the code for maximum convenience. We would comment out whatever ones we were not currently using to ensure that we didn’t lose the password somewhere. Then we would check it into our code repository, ensuring that everyone on our entire team could also see the secret. At the time, none of us had any idea that there was a better way to do it.

Fast-forward a few years, and I became a junior penetration tester. I learned that there were all sorts of other places where you could keep your secrets and that there were products you could buy to do all of it for you properly. I also

learned that storing such things in your code was a serious security no-no! I thought back to many of my previous habits as a developer and promised myself that whenever I find secrets in code, I will be patient and kind when I deliver the bad news.

Unfortunately, those of us who are looking for them continue to find secrets in code constantly. I found live banking passwords in 2021 during the *one* security assessment I performed that year. I don't even work as a pentester anymore, and although I'm skilled in this area, I'm not an expert. I was just doing a client a favor. What this means is that the message has not been delivered to software developers that there is a better and easier way to manage their secrets than storing them in the code.

For secret management, the first step is to scan for secrets in *all* the code for your entire organization, which is surprisingly easy and cheap to do. There are many secret scanners available on the Internet for free, and most of them use something similar to Linux's `grep` command, using regular expressions (REGEX) to match patterns that look like secrets. If there's high entropy and/or high randomness, it might be a secret. If there's a variable called `key`, `password`, or `secret`, and that variable is being assigned a value, it might be a secret. These tools can analyze huge amounts of code in very little time, with decent results. Go through all of it and take note of any legitimate secrets. If possible, tune your REGEX so it stops returning false positives and then save it.

NOTE There are paid secret-finding products that will use more advanced approaches for finding your secrets, as well as validating if the secrets are “live.” They will save you time and have higher accuracy but cost you money. Choose based on whatever you have more of: time or money.

Next, purchase a system to manage your secrets. I know, you are a software developer, and you can *make one*. Trust me—buy one. It is likely to cost less and is pretty much guaranteed to be more secure. Please do not make your own unless you work for a company that builds security products; even then, it will be a large undertaking.

Once you have acquired a secret management system, you will need to deploy it. You can hire whomever you bought it from to help you, or you can do it yourself. No matter who does it, ensure that you have a different vault/instance for each system. You should not save all the secrets for every application your organization maintains and builds into a single vault. If you did that, then if one app was compromised, it could authenticate to the vault and steal all the other apps' secrets. Each application or system needs to have its own vault.

Now to put things into your secret management system! Take every live secret you found, rotate it, and then check the brand-new secret into the vault. Do not save a copy of the secret somewhere else. Your app can authenticate and get the

secret, and so can any other apps you have set up that way. No human should ever be able to retrieve the secret, and it should not be saved anywhere else. If a human does “need a copy” of the secret for something, rotate it, use it in whatever way was required, and then save it into the vault and nowhere else. Examples of when you may need a copy of a secret would be if your secret has been leaked, spilled, or otherwise shared/ruined or if a new version of an API is released and requires you to create a new key.

Secrets can be accessed by your applications via code (programmatically), an API, and even in the CI/CD.

TIP Rotating a secret means generating a new secret and saving it to your secret management tool. Throw out the old one, and you’re done!

TIP Secret management systems that support multisigning keys (for example, Shamir’s secret-sharing algorithm²⁵) also benefit privacy and General Data Protection Regulation (GDPR) compliance by shredding/revoking keys. *Cryptographic shredding* is the practice of deleting the encryption key for data in order to “delete” the data (making it irretrievable and therefore unobtainable; this is considered equal to it being deleted).²⁶ This is a not-uncommon tactic for GDPR compliance in event-sourced systems: delete the key for a user to destroy all their personally identifiable information (PII). Shredding the key makes decrypting impossible and effectively renders the data irreversibly unavailable.

Password Management

In this section, we are going to cover passwords that are used to authenticate to your applications for your users. We are not covering password management for individuals; we will talk about how your application and database should manage and store the passwords that your users use to access your custom software.

Right now, all over the dark web, there are credentials for sale. Credentials are made up of one password and username that when used together can successfully access at least one system. Also right now, all over the planet, a large percentage of users are overwhelmed with how many passwords they are expected to remember and, as a result, practice something called *password reuse*. Password reuse means using the same username and the same password for several systems. As a result of so many people practicing password reuse, criminals know that if they buy a large set of leaked or stolen credentials, a decent number of them will authenticate properly on many sites on the Internet. They then use those stolen credentials to steal from unlucky users and often commit other crimes as well. Obviously, we want to protect our users from this terrible fate.

NOTE When criminals use several sets of credentials against the same site, trying to break into it, this is called a *credential stuffing attack*. Quite often, users add numbers to the end of their passwords, such as “Password1” or “Winter2025,” and criminals are aware of these trends. When the attackers adjust the password values, incrementing or decrementing the numbers on the end, this is called either a *password spraying attack* or a *rainbow credential stuffing attack*, depending on whom you ask. The *rainbow* refers to rainbow tables that are used to try to break hashing algorithms.

There are a few ways we can help solve this problem. One is to not use passwords at all. There are a few options for this, one of which is called *passwordless* (systems that use anything other than a password for authentication). An example of this would be having your network identity system “recognize” users when they attempt to access systems and silently let them in. A certificate in a device can also be used for passwordless, as well as fingerprints, facial recognition, and more. Unfortunately, many users don’t select this option when offered.

Another no-password option is *passkeys*, which use your biometrics (fingerprint, most often) to identify you. More information can be found at <https://passkeys.dev/>. Users are now starting to adopt passkeys more often. This is great news for security enthusiasts.

The key takeaway here is that the less friction there is for the user, the better. If you must use passwords, though, there are several things you need to know to ensure that your user’s credentials are never stolen and then misused.

The first thing we want to do is create a system where users’ passwords are hashed, not encrypted. Hashing is a one-way cryptographic method, meaning you can never* recover the original value. Encryption, on the other hand, can be followed by decryption (assuming you have the key and know the algorithm) to recover the original value. At this point, you might be thinking, “Why would I never want to get my user’s password back? Without a copy of my user’s password, how do I know if I should let them in or not?”

NOTE Never*: as of this writing, current computer capabilities are unable to reverse hashing algorithms. In a post-quantum computing world, this may change.

After a user has set up their password for the first time, when your system asks them to log in and they enter their password, it is hashed with the same algorithm you used when they set their password. Then you compare it to what you have saved. If they are the same, you let them in. If not, they got it wrong, and they can try again.

Now that we know we need to hash, not encrypt, the stored value, what’s next? Salting. *Salting* means adding a random value to your password (at the start or end) and then hashing it. You store both the random value and the hashed value in your database.²⁷ When users enter their password into your system to

authenticate, you add the salt to it, put it through the hashing algorithm, and then compare it to what you have stored.

TIP There are countless sites online with rainbow tables already created for you to help you crack spilled passwords; it's much easier than you think. Salting passwords is an absolute must!

An additional level of security you can add, on top of hashing and salting, is called *peppering*. (No, I am not kidding. Please feel free to make jokes about adding salt and pepper to your AppSec meal.) A cryptographic pepper is similar to a salt in that it is random and added to your password before hashing, but it is considered a secret and not stored alongside your password in the database. It can be stored in a secret management tool or another safe place, such as a hardware security module. Each organization has only one pepper value, which is used for every user. Keep it safe!

There are a few drawbacks to using a pepper. From an administrative viewpoint, it can be a lot of work. If you lose the pepper, you will have to force a password reset for 100 percent of your users. On top of this, eventually you will need to rotate your pepper, which is quite a lot of work, even if you plan it in advance. The use of a salt plus hashing is usually “secure enough” for most organizations, but peppers can be used for an even higher level of security assurance.

Communication Security (Cryptography and HTTPS Only)

When we transmit anything over the Internet, it should be encrypted. Gone are the days of very slow Internet and computers with very little RAM. Thanks to the Electronic Frontier Foundation (EFF), we can all get free certificates for any site from Let's Encrypt (<https://letsencrypt.org/>). This is why I, and many other security professionals, strongly encourage that we only serve web apps and APIs over the Internet via HTTPS. Every single request and response should be encrypted with the latest stable release of Transport Layer Security (TLS).

On top of this, it is important that we use products that perform the encryption for us. Unless you work at a place that creates encryption products, software developers should not attempt to write their own encryption software. It is *very hard*, which is why we have experts that do this for us. I realize I probably sound like a broken record by now.

HOMEMADE “ENCRYPTION”

Bob remembers going to something called Capture the Flag (CTF) with some friends he met at an OWASP meetup. The first challenge was to “decrypt” some homemade encryption. The developer of the challenge had base64-encoded the password twice rather than performing real encryption. Bob saw how the people on his team

copied the value into a website that decoded it for them; it took under a minute! Oh my gosh! Bob had no idea it was so obvious to others when values were not actually encrypted. . . . He made a mental note to never let his developers write their own code for such an important task.

TIP Even if you build your own real encryption system (not just encoding), hackers can still get into it! Often, at CTF contests, participants will be set against “homemade encryption” systems, and they *still* manage to break into them. Encryption is a difficult problem; please leave it to the experts.

Protecting Sensitive Data

A common security issue with any type of application or computer system is when it reveals sensitive information that it should not. When this situation happens, it is often referred to as *sensitive data exposure*, a *data breach* (when the data has been stolen), or a *data spill* (when the data was leaked by accident). No matter what you call it, it’s dangerous and can cause serious harm to individuals, businesses, and governments.

Sensitive data can be any of the following, but this is not an exhaustive list:

- Personally identifiable information (PII)—anything that could be used to steal or reveal your identity, such as your full name, date of birth, home address, social insurance number, driver’s license number or image, where you work, etc.
- Health information, such as whether you have a health condition, how your health condition affects you, if you are on certain medications, etc.
- Intellectual property, such as manuscripts for books, movies, plays, copies of art (visual art, music, etc.), or anything else that can be legally copyrighted.
- Business intelligence and other business secrets, such as strategic direction, designs, plans, etc., that could harm a business, should it be revealed publicly or obtained by competitors.
- Classified, secret, or top-secret government information or data, such as counterterrorism plans, governmental policies that are not yet released, investigations, etc.
- Customer information, such as what they purchased, purchase dates, how much money they spent, services they used, etc. Generally, customers want privacy from the organizations they patronize, even if they are just buying a pair of pants.

- Anything that, if it was released, would cause embarrassment, loss of funds, physical danger, or any other damage.

The first step in avoiding sharing sensitive data by mistake is not collecting data in the first place that we do not need to collect (also known as *data minimization*). If we are building a super-amazing “to-do” app, do we really need the user’s date of birth, home address, and full name? We probably don’t. If we avoid collecting information we do not need, this reduces our attack surface by minimizing the amount of data we need to protect.

The second step for avoiding sharing data in error is to classify and label the data so everyone who accesses it knows its sensitivity. As a result, it will be treated and protected according to its classification.

Classifying data means figuring out what level of sensitivity it has. Perhaps something is classified as “unclassified” (United States) or “public” (Canada); this means it can be released to the public, will cause no damage, and can be shared freely. That said, if something is marked “classified” (United States) or “protected” (Canada, levels A, B, or C), someone using it will become immediately aware of how they should treat it.

Labeling data means just that: attaching some sort of label to the data. This can be an extra field in your database, writing onto a CD/DVD/hard drive, labeling your cloud storage container, etc. The main point of labeling is to make it as obvious as possible to others.

TOP-SECRET DESTRUCTION

Alice used to work in a top-secret bunker very early in her career. Although they labeled CDs and DVDs with “top secret,” once the CDs and DVDs were no longer needed, Alice used to chop them up with a paper cutter and put them in a safe. Once a week, they were sent to be burned. She also used to work somewhere that used a “burn box” rather than shredding documents. Sometimes the data is so valuable or sensitive that physically destroying it is the only way to ensure that it is kept safe.

The next step in protecting sensitive data is following whatever policies or instructions your organization has set out for you. If you do not have a policy to follow or if your policy is limited or otherwise “not good enough” for you, please review the following items.

Sensitive data protection highlights:

- Ensure that you are following any regulation, legislation, compliance, policy(s), or rules that apply to your data, such as GDPR (Europe), The Personal Information Protection and Electronic Documents Act (PIPEDA—Canada), or the California Consumer Privacy Act (CCPA—United States).

- Do not log sensitive data. This means complete credit card numbers; American Social Security numbers (SSNs); Canadian social insurance numbers (SINs); a user's name, DOB, and home address; etc. When in doubt, ask the security team.
- Encrypt *all data* (sensitive or not) in transit and at rest. If the data is very sensitive, ensure that you wipe the memory of the client machine once it is no longer being accessed.
- Only store data for as long as you are required to do so. Set it to auto-delete on that date, if possible. For instance, if you must keep a certain type of data for 7 years, set it to auto-delete itself once it hits that age. Build these features into the system from the start.
- Do not put sensitive data into hidden fields, URL parameters, or any other place it could be seen by the end user. Even if you encrypt the data, those are still not appropriate places to store or send sensitive data.
- Do not cache sensitive data, ever.
- Verify a user's identity (authentication) and that they are allowed access (authorization) before granting access, and then reverify their authorization for every additional request (every new page or record). Reauthenticate users at a regular cadence on very sensitive systems (this can be anywhere from every 15 minutes to every few days; ask your AppSec team).
- For very sensitive data, sometimes a tokenization system can be used, which changes the data into encrypted tokens and only reveals the real values when necessary. The values are encrypted until the moment you need them (often called "just in time"). Software developers will implement this with the assistance of your security team or a third-party contractor and generally will not be expected to do this without expert assistance.
- Log every access request for sensitive data, and create security alerts for repeated failed attempts or anything that looks like potentially dangerous or unusual user behavior.
- Monitor your systems for anomalies or anything that may be malicious. Generally, the security team will implement such a system, but they may ask for your help.
- Follow the rest of the secure coding advice in this book.
- Work somewhere that takes the security of your information systems seriously. Choosing the place you work based on how important information security is and the presence of a strong security culture is likely the best way for you to grow into an amazing, secure coding professional.

Security Headers

An HTTP header is a field of an HTTP request or response that passes additional context and metadata about the request or response.

—Mozilla²⁸

HTTP headers communicate to the browser how the developer wants it to treat requests and responses for your web app.

—Tanya Janca (*that's me!*)

In my first book, *Alice and Bob Learn Application Security*, I covered all the security-related HTTP headers that existed at that time (late 2020). Since then, a few new ones have been released, along with new features on already-existing headers. For the ones that were in my first book, I will give a brief description here, and you can either look them up online or in my book if you want more detailed information.

New Security Header Features

These new features were created to allow us to be more intentional with how we want the browser to handle security. They give for the developer and application more context and more control, which allows for safer and more robust applications.

Fetch Metadata Request Headers

These new request headers allow the browser to add more context to a request to help the server decide whether it's a good idea to perform the request. These decisions can be done on the server or "lifted up" to the content delivery network (CDN) or reverse proxy level (a Web App Firewall, WAF or Run-time Self Protection, RASP). This type of header exists to prevent potentially malicious actions taken on behalf of the user that are requested from outside sites and/or scripts (also known as cross-site). We add information about the source and context of the request using these headers to give the server the information it needs to make a good decision:

- `Sec-Fetch-Site`: Specifies which site generated the request
- `Sec-Fetch-Mode`: Specifies the intended request mode or type of request
- `Sec-Fetch-User`: Specifies if the request was initiated by a user action
- `Sec-Fetch-Dest`: Specifies the intended request destination

To read more, visit <https://w3c.github.io/webappsec-fetch-metadata>.²⁹

Content Security Policy Header

This one is not new, but there are new features within it that need mentioning: `strict-dynamic` and `trusted-types`.³⁰ The rest of this header will be explained later in this section.

Strict-Dynamic

Content security policy (CSP) allows you to list your application's resources and scripts to prevent malicious actors from adding their own. Often, people used a nonce (a random value used to identify the script is authorized to run), but it was quite complex and hard to get right. When using a nonce with CSP, if you do not control some of the third-party scripts, or your scripts create child scripts, you will not be able to add a nonce, ruining your best-laid plans. With the introduction of the `strict-dynamic` option, only scripts in the response body require a nonce, and it allows scripts to create their own children scripts (without the need for their own nonce). Although you still need to do some work, this reduces the amount of effort required to refactor a web application to use CSP. For a detailed guide on how to use CSP, visit <https://csp.withgoogle.com>,³¹ and to test if your CSP is implemented correctly, visit <https://csp-evaluator.withgoogle.com>.³²

Trusted-Types

DOM-based XSS is the most difficult type of XSS to avoid, find, and remediate. The `trusted-types` option allows the developer to create variable types that are safe from XSS and then use CSP to enforce that only those trusted types are allowed to be used within your application. When `trusted-types` is enforced, your application may only use those types; no other types of variables are allowed.

To create a trusted type, you create a trusted-type policy that includes validation. You then use the policy to create trusted-type objects; and finally, you enforce the `trusted-types` policy using your CSP header. To learn more about trusted types, visit <https://github.com/w3c/trusted-types>.³³

Special thanks to Lukas Weichselbaum, his team at Google, Michael Koppmann, and the Sec4Dev conference 2022 (Vienna, Austria) for teaching me about these two new innovations in CSP and trusted types. Follow Lukas for more browser innovations and Chrome security feature updates.

Security Headers Previously Covered

What follows is a very brief explanation of the various security headers available to developers of web applications, which were covered in more detail in my first book, *Alice and Bob Learn Application Security*.³⁷

Content-Security-Policy Header

The Content-Security-Policy header requires you to list all resources (scripts, images, etc.) that you want to use in your application that are not from your own domain. It blocks XSS and framing. This is the most difficult header to implement, but it is a very powerful form of defense. Feel free to ask for help from the AppSec team.

This is a closed implementation of CSP that blocks everything:

```
Content-Security-Policy: default-src 'self'; block-all-mixed-content;
```

If you plan to include anything from other domains in your app, it *must* be declared in Content-Security-Policy. If it's not declared, it won't work. You can do more configuration than is shown here, but these are most likely what you will need:

- `object-src` = plugins (valid sources for `<object>`, `<embed>`, and `<applet>` elements)
- `style-src` = styles (CSS—Cascading Style Sheets)
- `img-src` = images
- `media-src` = video and audio
- `frame-src` = frames
- `font-src` = fonts
- `plugin-types` = limits types of plugins that can be run

HTTP Strict-Transport-Security

Strict-Transport-Security (HSTS) enforces the use of HTTPS only (redirects from HTTP to HTTPS). This example is set for one year:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

X-Frame-Options

This blocks sites from framing your site. It works for older browsers that don't support the CSP header:

```
X-Frame-Options: SAMEORIGIN (you can frame yourself/from within
your domain)
X-Frame-Options: DENY (no frames at all)
```

X-Content-Type-Options

This blocks sniffing of content types and forces you to declare types. It also stops sniffing attacks:

```
X-Content-Type-Options: nosniff
```

Permissions Policy

This disallows or allows various HTML 5 features. The default is to disallow unless required:

```
Permissions-Policy: camera 'none'; microphone 'none'; speaker 'none';  
vibrate 'none'; geolocation 'none'; accelerometer 'none'; ambient-  
light-sensor 'none'; autoplay 'none'; encrypted-media 'none'; gyroscope  
'none'; magnetometer 'none'; midi 'none'; payment 'none'; picture-in-  
picture 'none'; usb 'none'; vr 'none'; fullscreen *
```

Expect-CT

This security header has been deprecated since my first book was published. Do not use it.

Referrer-Policy

Use this to avoid sharing previous pages browsed by users and for user privacy:

```
Referrer-Policy: origin
```

Public Key Pinning Extension for HTTP (HPKP)

Do not use. This header is deprecated and dangerous to use.³⁴

X-XSS-Protection

This head is deprecated and can cause a small amount of risk. It is preferred that you do not use it.³⁴

TIP Use every possible security header. Think of them like seatbelts in your car: they only help when there's an emergency, but gosh, you're happy you used them when something goes wrong. Once you get used to using them, it's a breeze, and they might save you (or your app!) someday.

More New Headers

Although these security headers came out in 2021, I am calling them “new” because almost no one is using them, meaning they are mostly like *new to you*. Security headers and other browser security features change over time to address new types of attacks and risks. The headers in this section (except CORS, which existed before 2021) were all created to help prevent side-channel attacks, such as specter and meltdown.

Same-Origin Policy

The `same-origin` policy is a browser security feature that restricts how documents and scripts on one origin can interact with resources on another origin.³⁵ It was created to try to stop web applications from attacking each other. It is included and enabled by default in modern web browsers. This policy disallows scripts to be run in an application if they come from a different origin. *Origin* is defined by URI scheme, domain, and port number. If all three are the same, the script can run. If any of those three are different, it will not run unless it's listed in your CORS.³⁶ The `same-origin` policy will not prevent loading of other page resources, only scripts.

As a security person, I'm very pleased this is enabled by default!

To understand the rules for configuring the following three security headers, it helps to understand a few terms they use in their settings.

Same-site means everything within the same domain, including subdomains. For instance, `https://shehackspurple.ca/` and `https://newsletter.shehackspurple.ca/` are both same-site but not same-origin.

Origin refers to the domains, scheme, and ports. If all three of them are the same for two URLs, then both URLs have the same origin (same port, subdomain/domain, and scheme). For example, `https://shehackspurple.ca/blogs/article1` and `https://shehackspurple.ca/blogs/article2` are both same-site and same-origin.

However, `https://shehackspurple.ca/blogs/article1` and `https://newsletter.shehackspurple.ca/article1` are same-site but different origin (due to their different subdomains). If you want to share resources between these two sites, you need to either use none of the following security headers (and leave yourself at risk of other sites also having access to your resources, plus side-channel and other types of attacks) *or* use the following security headers and explicitly give permission to share.

COEP: Cross-Origin Embedder Policy

This header dictates how cross-origin resources are allowed to be embedded into a document. All cross-origin resources must be explicitly listed in CORS or CORP (more information shortly), or they will be blocked.

`unsafe-none` is the default config value, which allows you to fetch resources from anywhere without the need to explicitly state them. It's called `unsafe` for a reason!

`require-corp` allows resources from within the same origin (your domain) and any that are explicitly marked as "loadable" from another origin (you must list them in the CORS or CORP header to mark them as loadable).

```
Cross-Origin-Embedder-Policy: (unsafe-none | require-corp);
report-to="default"
```

* All of these headers support reporting via the reporting API. If you're using REPORT-URI or using the API directly, you can see how your security headers are protecting your app.

COOP: Cross-Origin Opener Policy

This header prevents other processes from looking at your memory by breaking your app out of the Browsing Context Group. It does this to isolate your app and its information. This header ensures that you are not sharing a context with any other page, including potentially hostile pages. There are levels you can configure to define your co-existence with other pages inside the COOP.

Same-origin means your app shares context (and therefore potential access to the data stored in memory) with other pages that are in your origin *and* also have their COOP header set to the same origin. This means when you open a new window, if it's not in the same origin as your app, it's opened into a new browsing context group (protecting your app from sharing memory and anything else stored or accessible from your origin).

If you want to share your origin with a popup, you can do this explicitly by using Same- origin-allow-popups.

The last option for this header is `unsafe-none`, and this is the default way the application will act if you do not have this header set. It allows your page to be in the same browsing context group as all the other pages that do not have the COOP header set. It's called `unsafe` for a reason!

```
Cross-Origin-Opener-Policy: (same-origin | same-origin-allow-popups |  
unsafe-none); report-to="default"
```

CORP: Cross-Origin Resource Policy

This requires that all resources be explicitly permitted (listed) to be loaded cross-origin with a CORP header. You can also define how those assets are allowed to be loaded.

The `same-site` setting is the strictest, with different subdomains being unable to load resources from each other.

`Same-origin` applies to most sites, as it shares resources within your site and all the subdomains.

`Cross-origin` allows everything and is advisable only if your site is required to serve resources for others. A content delivery network would want this setting. Most other websites would not want this setting.

```
Cross-Origin-Resource-Policy: (same-site|same-origin|cross-origin)
```

CORS: Cross-Origin Resource Sharing

CORS has existed for many years, and no one considers it "new" by any means. That said, it relates to all the other "new" (released in 2021) security headers.

CORS tells servers that host assets which origins (domain, port, and scheme) are permitted to load those assets. CORS can be used to prevent others from loading your assets, such as blocking people from showing your copyrighted images on their sites.

CORS breaks up how to load the assets based on the type of resource requested. It will either just do what you asked (for instance, if loading an image) or send an HTTP Options request (to explain the complex part: for instance, if you are using a less-common HTTP method as part of your request, such as delete, trace, or head), and then it will send your intended/original request afterward. The first request, with `http options`, is called a *preflight* request and only happens sometimes (when your request is “complex”).

```
Access-Control-Allow-Origin: domain-this-resource-can-be-shared-with
```

CORB: Cross-Origin Read Blocking

The CORB security header is on by default in Chrome browser versions 68 and above. Other browsers are currently in various states of rollout. It is best known in developer circles from the unfortunate few who have received the dreaded “blocked resource” error message.

```
Cross-Origin Read Blocking (CORB) blocked cross-origin response MY URL
with MIME type application/json.
```

This header blocks requests for data resources to be asked for as images. This means your application cannot be tricked into accepting JSON, HTML, or XML as an image (when you were expecting an image), which is sometimes possible without this security header. It blocks potentially malicious content from loading at all if it mismatches what you are expecting, ensuring that the attack is unable to self-start.

This header protects the following types of content: HTML, JSON, and XML.

For this header to work properly, you need to configure the cross-origin headers CORS and CORP, plus set `x-content-type-options` to `no-sniff`. To protect these three data types, you must have CORS set to prevent most or all origins from loading, meaning setting `allow-origin` to anything but `*`. Please don’t use `*`; it not only negates some of your other protections but also opens these three data types to side-channel attacks.

```
X-Content-Type-Options: nosniff
```

```
AND
```

```
Access-Control-Allow-Origin: URL-of-desired-origin | definitely-not-*
```

* In earlier versions of Chrome, CORB is either activated together with the Site Isolation feature or can be manually activated by launching Chrome with the following `cmdline` flag:

```
--enable-features=CrossSiteDocumentBlockingAlways
```

Secure Cookies

What follows is a very brief explanation of the various security settings for cookies, which were covered in more detail in my first book.³⁷ Please use all of them.

Secure flag: ensures that cookies are only sent over HTTPS:

```
Set-Cookie: Secure; (plus all of your other settings)
```

HTTPOnly flag: blocks JavaScript from accessing your cookie / reduces harm from XSS:

```
Set-Cookie: HttpOnly; (plus all of your other settings)
```

Expires: ends cookie persistence / deletes old cookies. The following is for January 1, 2021:

```
Set-Cookie: Expires=Mon, 1st Jan 2021 00:00:00 GMT; (plus all of your other settings)
```

Max-age: ends cookie persistence / deletes old cookies. The following is for 1 hour:

```
Set-Cookie: Max-Age=3600; (plus all of your other settings)
```

NOTE Use **Expires** or **Max-age**, not both.

Domain: if you want to share your cookies with other domains:

```
Set-Cookie: Domain=app.NOTshehackspurple.ca; (plus all of your other settings)
```

Path: limits cookie access to only certain parts/paths of your domain:

```
Set-Cookie: path=/YourApplicationsPath; (plus all of your other settings)
```

Same-site: disallows cookies from other domains to be used by your app:

```
Set-Cookie: same-site=Strict; (plus all of your other settings)
```

Preferred option

```
Set-Cookie: same-site=Lax; //only blocks POST requests, allows links, XSS blocking
```

TIP Classify the data in your cookies so you know if they contain sensitive data. Add labels if possible, and document them. The more sensitive the data in the cookie, the more careful you should be with it and the shorter the amount of time it should persist.

Error Handling

If you're reading this and you are a software developer, you already know that you need to "handle" your application errors. "Handling" a program error means having code that reacts when a system or program error occurs, ideally with a message to the user, logging of the event, and, if necessary, triggering an alert to the security and/or development team(s).

If we do not handle our errors, our application can fall into an unknown state, which can result in a crash, freeze, system errors, strange behavior, and/or vulnerability. As a secure coding developer, you want to avoid having any of your applications fall into an unknown state, as it is the place that every malicious actor aims for. It is when our application performs in unexpected ways, which can lead us to data breaches, privilege escalation, and more.

For instance, if we do not "catch" an error, our application may crash. If we have a case statement with no "else" or catchall statement at the end, but none of the case options are true, the application doesn't know what to do next:

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  WHEN conditionN THEN result
  ELSE result
END;
```

If there is no `ELSE` section or there is no code in the `ELSE` section, the application may not know what to do next. This can lead to unexpected outcomes.

Writing error handling for every function or block of code may be time-consuming, but it can help you avoid problems when you get to the testing phase or, even worse, after you release to production. Adding a global exception handler will ensure that if you missed something, your application still has a safety net. No matter how you design your error handling, it is very important that you do intense testing to ensure that you haven't missed anything.

If you are partway through a transaction when an error occurs, always roll back the transaction and start again. This is another situation where vulnerabilities may lie: partially completed transactions.

Finally, when handling errors, never share the system-generated error with the user; always write your own user error messages. Errors generated by a database or web server may contain information that would be helpful to an attacker, such as the version, patch level, syntax, or other system information. Instead, write your own custom message with as little technical detail as possible—just enough for the user to understand what they need to do next.

TIP Always turn off `DEBUG` mode when publishing to production! This can help you avoid accidentally splashing a stack trace or other embarrassing system errors on the screen.

Last week, Bob was using some software, and it crashed. It showed him a system error, and it made him remember the CTF competition he went to years ago. His CTF team found a way to make the application give them SQL server error messages when they put what looked like a bunch of junk to him into the login password field. One of the people on the team was a wiz with SQL, and he used the error message to change his SQL injection attack. He said the error messages were feedback from the server, correcting his bad syntax. Thanks to the server error messages, they were able to successfully bypass the login screen and get the flag! Although Bob was happy they were winning the competition, he was secretly worried about the apps back at the office. From then on, he made it very clear to his development teams that they could never show a system-generated error message to users; it was too dangerous to share such information.

In this chapter, we covered the beginning of the secure coding journey and created a groundwork to build from for the rest of this book. We discussed following a secure SDLC, using modern frameworks and their security features, as well as the value those decisions will provide. We covered several types of foundational security controls, including input validation, output encoding, authentication, authorization, and session management. If you implemented all the learnings in the book so far, you would be creating some decent applications! But there's more to learn to create even more robust, reliable, and secure software. Onward!

Chapter Exercises

You can find the answers to these questions in Appendix B, "Answer Keys."

1. Input to an API that comes directly from your web app does not need to be validated. True or false?
2. How many times does someone have to sign in for SSO?
3. In what order do you do authorization and authentication? Why is the order important?
4. Name three types of secrets that you would store in a secret management tool.
5. Same-Origin policy is enabled in the web browser by default in modern web browsers. True or false?

6. `Cross-Origin-Opener` policy is enabled in the web browser by default in modern web browsers. True or false?
7. Give two reasons why you should not print system error messages directly to the screen.
8. Give two reasons why you need to catch our errors and handle them in our applications.

Improving

Now that you have begun your journey to becoming a secure coder, we will improve on what you have learned so far. We will go into privacy, locking down the technologies you are likely to use (databases, files, and memory), and how to make decisions that can prevent future attacks from happening.

In this chapter, we will prepare to protect ourselves from “average attackers,” not extremely advanced ones (such as nation-states). When working with clients, as a security professional, I always want to know what “security posture” they are aiming for. How secure do they need to be, *really*? Usually, at the start of the conversation, they say they want to be as secure as possible, but unfortunately, that can be extraordinarily expensive and cumbersome. We don’t want to spend \$1 million protecting something worth only a few thousand dollars. We also don’t want to have something locked down so tight that it is barely usable unless we absolutely must ensure that level of safety is required. Usually, after a few minutes we can come up with an understanding of what they want and need and then plan how to achieve this level of assurance.

When you are creating applications for your employer, ideally your security team will outline the security requirements for your projects, systems, and data. If they do, you can use the information in this book to help you meet their requirements. However, if this is not the case, for whatever reason, it might be up to you to decide how much security is “enough.” Please try to use the information you learn here to do your best to protect the systems you build and the data they house.

Database Security

In computer science, a database is an organized collection of data stored electronically in a computer system. You can use a database management system (DBMS) to store, retrieve, and edit data. The word *database* is often referring to any DBMS, database system, or an entire application associated with the database.

—Wikipedia³⁸

Those of us who work in information security (also known as InfoSec) are charged with protecting the (physical and digital) information and information systems of our organizations. Information technology (IT) security protects just the digital assets, and generally in this book, we are referring to IT security, not the protection of physical assets. This scope includes the entire IT department, all the digital documents, networks, applications, and their associated databases. Every bit of digital information is under our charge, and we need *your* help to ensure that it is protected properly. In this section, we will talk about database security.

TIP *Hardening* means locking down a system. It means using secure configuration settings; ensuring that your patches are up to date; disabling all protocols, services, and functionalities not required for the system's operations; and anything else that will make the system tough and secure. Most systems come with a hardening guide, which you can ask your vendor for. Follow the guide for the best results!

Four Perspectives for Protecting Databases

For starters, databases are servers, which live on networks, host database software, *and* house data.

Let's talk about the server aspect first. We need to harden and scan our database server, ensuring that all security patches are up to date and that the configuration follows security best practices. Generally, a hardening guide is provided by the vendor that supplied your database software. Then we want to scan it regularly with a vulnerability assessment tool (often called a VA scanner) to ensure that we have done a good job and continue to do so. This responsibility generally falls on the security team.

TIP You can use a managed database from one of the public cloud providers to avoid having to patch and maintain your database server.

The next thing to look at is the network this server sits on. Is it public? I hope not! Does it have different “zones” that are protected by firewalls? Does it follow a “zero trust” model? This is generally something the network team and security team work on together.

Now we're down to the database software level, administered by the database administrators (DBAs), often with advice from the security team (the security team has their fingers in everything, don't they?). The database software can also be scanned with security tools to ensure that it has the proper hardening settings applied to it. You can read more about how to test, scan, and protect your database server software in *Alice and Bob Learn Application Security*, in Chapter 6.³⁹

WINNING ROLLBACKS

Bob remembers when he first became a programmer for the Canadian Federal Government. In his first job, they insisted that he create a rollback script for every script he submitted to the DBAs. He had to practice running his script on the dev database and then rolling it back and checking that the data was a perfect copy of what it had been before. Although Bob thought it was overkill at the time, the DBAs at that office were some of the best he ever worked with. He respected them for taking their jobs very seriously. Bob kept these practices up throughout his programming career, and it saved his bacon more than once!

The last part is for you and the security team. Protecting the data inside the database requires classifying and labeling the data and then treating it according to its classification level. These rules for handling sensitive data cover how to access, store, manipulate, and back up the data. These rules may also dictate that you *not* collect certain data, segregate some of it, or otherwise treat it differently than the rest due to its classification. They will cover how often you back up your data, when to practice rolling it back (also known as *restoring*), and even when to dispose of the data. There's very little point in doing data backups if you do not practice rolling them back at least once. The rules will also cover whether you should store your sensitive data in a different table, in an encrypted or tokenized format, and more.

Other important ways to protect your data include the following:

- Your data should be encrypted in transit (being sent somewhere) and at rest (when stored somewhere). Use the most recent version of TLS that is supported, and follow the policy from your workplace for key length and acceptable algorithms to use for encryption. Note: please do not attempt to write your own encryption methods.
- Add logging and monitoring such that queries to the DB can be monitored.
- Create least-privilege access roles: e.g., read-only roles where applicable or roles that only grant access to specific tables.
- The account you use to access the database should have a long, unique password. If possible, turn on two-factor authentication (usually called either 2FA or MFA).

- Some cloud providers allow for temporary credentials without passwords (e.g., identity and access management [IAM] roles); this is another secure option.
- When your applications and other IT systems access the database, they should use a service account (an identity on the network that is unique for that system), not an individual's account. The passwords for this account should be managed via a secret management tool and accessed programmatically rather than having human beings involved.
- Do not save information from the database outside of the database. Do not make copies for yourself or others unless it is a project requirement. Please never make copies on your local machine.
- Do not use a copy of production data for any other environment (such as dev, QA, UAT, etc.). Production data should not be used for quality assurance testing, development, or anything other than production use. Instead, use a masked version of the data or generate sample data for lower-level environments.
- Never share your passwords or accounts with anyone else for any reason.
- Do not view production data in the database. Developers shouldn't be viewing customer data unless it's required as part of your job. This is about privacy, not about security.
- Do not leave your computer unlocked, connected to the database, and then walk away from your desk. This should be obvious, but it is mentioned for a reason.

TIP *Masking data means mixing it all up. It's the same information, but every field in a record is changed to information from another record. If a record was previously*

First Name = Tanya, Last Name = Janca, DOB = April 1, 1980

then after it's masked, it could look like

First Name = Tanja, Last Name = Reznor, DOB = Jan 1, 1999

It's no longer a real person's record, but you can still test the system, as the data is not complete garbage: the formatting is still perfect. But it's also completely inaccurate, which is what we want in order to avoid anyone using the real production data for less-than-honest purposes.

TIP When masking data, it is crucial that the data cannot be reverse-engineered to identify any individual, which means keeping any personal identifiable information (PII) in place could be identifying, depending on the dataset. Ensure that you change every single field, not just some of them.

IF ONLY WE HAD PRACTICED OUR ROLLBACKS

Years ago, there was a big network outage at the company where Alice worked, and everyone was sent home for the day. With no computer network or internet access, how could they work? The next day, the IT department reported that due to some sort of “glitch,” they had lost three days of work for the entire company! The head of IT told Alice it would take about a month to restore the three days’ worth of changes from their backups onto everyone’s systems and shared folders. Alice was outraged! She asked why it would take so long, and the head of IT told her they had never even tried restoring from backup before and had never practiced rolling back. They were starting from scratch and were unsure if it would even work. Alice was livid. She asked, “Why bother doing backups if you can’t restore from them? Why bother paying these people to make backups that don’t work? Why not lay them all off and save some money?” Alice was not very popular with the IT department after making statements like that. . .

BOB’S BRIBES

Bob’s programmers used to have to submit a ticket with their SQL scripts to the DBA team to have them run on production. Sometimes it took a long time, and he would see the DBAs outside taking a break and got really frustrated with the wait. One day, Bob was eating cookies in the lunchroom. A DBA said, “Wow, I wish I had cookies,” so Bob shared. Then his ticket got put at the front of the line and executed right away. Very quickly, Bob learned to keep a box of cookies in his desk so he could get his scripts run in hours instead of days, even though bribery isn’t “the right thing to do.”

File Management

Your files are a target for malicious actors. Files on your web server, on other servers in your network, on USB keys, on backup drives, on your computer, in “the cloud,” and even files saved into your database. Images, scripts, letters, emails, videos, the manuscript for your next book. You name it, and likely someone out there would like to steal it, change it, and/or ransom it back to you.

When we create software, especially software that is internet-connected, this is a potential attack vector. Even software that does not give direct access to a file, if programmed improperly, could unintentionally create a path on your network to files that are inadequately protected. In this section, we will discuss how to protect files that are part of your applications or on your web servers. We will assume that your security team will handle protecting the files that are not associated with or accessible by the software that you build.

NOTE Special note on your code: Your source code has incredible value, and you need to protect it, too! Always check your changes into version control (also known as a code repository or “repo”), and follow any processes put forth by your organization in this regard. Assuming this is private, proprietary (closed source) code you were paid to write for work, do not copy it to other places, such as a public code repo, your home computer, etc. This is intellectual property that your company’s competitors would likely be overjoyed to find a copy of from somewhere it should have never been.

First, you must use some sort of access management for your files if they are going to be on your web, file, or managed server, or accessed by your web app or API. Your web server and file server(s) should have all files marked as “read-only,” and they should have no execution privileges. Ideally, you also have application control tooling (previously known as application whitelisting) and file integrity monitoring (FIM) installed on your web server and file servers. Ask your security team about them!

When you save any files from a user (for instance, a profile picture for a user), ideally you will not save it to the web server; you will save it to a file server, which has a firewall around it (in case the file turns out to be dangerous). Before saving it to the file server, you should scan it to see if it is malware or looks harmful and, if so, reject it. On top of that, you need to have your application choose the file name and where it is saved and encrypt it. Do not let the user choose the name, location, type (you should have a list of approved file extensions, and the rest are rejected), or any other details about the file. The file should have some sort of labeling to specify its sensitivity. You should also restrict the size of the file and scan it to make sure it is the type of file the user told you it was.⁴⁰ You can learn a lot more about how to secure uploaded files on the OWASP File Upload Cheat Sheet (https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html).

NOTE The Communications Security Establishment of Canada (CSE) created and open-source a free tool to check uploaded files for malware; it is called AssemblyLine. It does not share information back to the Canadian Government or any other organization. Get it here: <https://cyber.gc.ca/en/assemblyline>.

On top of all this, your security team should monitor file access, ensure that regular backups are done, that the documents are encrypted in transit and at rest,⁴² only allow user accounts that have authenticated to the system to access files (authN and authZ), and ensure that files can’t be copied off the server. Ideally, they have a file access management system to handle all of those requirements. Occasional auditing will also increase your security assurance, as it may find errors or anomalies.

File Uploads

File uploads were covered in great detail in *Alice and Bob Learn Application Security*, Chapter 4, “Secure Code.” That said, it is an extraordinarily high-risk activity, so let’s talk about it again!

When a piece of software allows you to upload a file, the user could upload *anything*. Malware, a zip bomb (a file that unzips another zip file, recursively, until it crashes your server), illegal images, copywritten material, and more. If you authenticate and authorize your users first (have them log in to your software), they are significantly less likely to upload something malicious, but it still happens. You need to take precautions when allowing file uploads.

The easiest way to do this safely is to use a third-party plugin/library/tool that is well-respected and has been thoroughly tested for security. If you search online or scan it with a software composition analysis (SCA) tool, you can often tell whether a library already has known vulnerabilities in it, just as you should check all the libraries you use in your apps (we will dive into this in Section 3 of the book).

If you *must* write your own, though, please follow this advice:

- Ideally, restrict it to only certain types of files. If possible, avoid PDF and EXE; you will have fewer issues if you do.
- Verify the type (do not use the extension to do this; read the file headers) and file size before you save it.
- Scan the file for malware before saving. Consider using AssemblyLine (<https://cyber.gc.ca/en/assemblyline>).
- Don’t let the user set the path where the file will be stored. You must be in charge of that, not the user.
- Rename the file; do not keep the user’s file name.
- Save it to a file server, not the web server/container/PaaS.
- Ensure that the server you save it to does not have execution privileges, is backed up regularly and monitored, and has a firewall around it (keeping bad stuff in).
- The backups should be stored on a read-only server as well, with similar firewalling.
- Monitor the file server and its backup for inappropriate access or anomalous behavior.
- Learn more in Chapter 2 of *Alice and Bob Learn Application Security*, “File Uploads,” and the OWASP File Upload Cheat Sheet. OWASP Cheat Sheets are always a great place to start!

Your Source Code

As mentioned previously, you need to protect your source code as well. People may steal it, misplace it, or otherwise leak your intellectual property. More than once, when performing AppSec at various companies, I have found code in public GitHub repos, code for which the only copy is stored on someone's personal computer (which got erased), code being edited live on the production server (with no other copy), or, even worse, source code that was never recovered, and all they had was the binary file.

To protect your source code, you need to do most of the same things you do to protect files: ensure that you always check your code into version control, set up access control, do backups, monitor access to your code, and (if possible) have alerts for strange behavior. The security team can do even more, such as monitor your whole network to see if data is leaving in strange ways (also known as *exfiltration*), set up canary tokens (fake files that “call home” if they are stolen), pay services to report to you if they see your source code has been stolen, and a lot more.

TIP If you are protecting mobile source code, you will also want to minimize and obfuscate it to protect against reverse engineering. The code for mobile apps runs on the user's device, which they have control over, and not every user will respect your wishes to stay out of your source code. More in Chapter 5!

TIP If you are charged with securing an application for which you only have the binary (no source code), you still have options! You can use a first-generation static analysis tool (they do symbolic execution, as opposed to flow analysis and antipattern matching), also known as SAST, to evaluate how dangerous it is, then put a Web App Firewall (WAF) in front of it (to block bad input), and network segmentation around it (sort of like a moat around a castle). This will protect it from obvious attacks, reduce any damage it could cause to the rest of the network if one day it is successfully attacked, and allow you to assess what level of risk it poses to your organization.

LOST WORK

When Alice supervised a development team, another team leader, Chad, had 11 consultants reporting to him, building custom forms for SharePoint. Alice's team had set up a team foundation server (TFS) for all the code to be stored for Chad's team. Chad's team was off the network, having created their own network to avoid following the company's IT policies (also known as *shadow IT*). Alice said that was not okay, but Chad did it anyway. Chad said he would back up the systems every night and “Don't worry; it will be fine.” One day, Alice saw Grace (the other dev team leader) walk by her office

carrying a big server. “Excuse me, Grace, what is that?” Grace said it was the TFS server Alice had set up for Chad. Grace said it had been offline for a few weeks, and because no one was using it, she had formatted it! Alice ran to see Chad, who told her it had been offline for weeks but hadn’t wanted to bother her with it. She asked when the last backup had been, and Chad admitted that he “never got around to doing a backup.” Although they were able to recover some parts from local machine copies, most of the work done by the 11 consultants for 5 months had been lost. Alice was outraged!

Memory Management (Buffer, Stack, String, and Integer Overflows)

At the first OWASP meetup I ever went to, we did a buffer overflow workshop.⁴³ I had been out of college for 14 years and had forgotten all about how the stack worked and memory management. In modern languages and frameworks, memory is managed *for you*, but when I first learned to program, I learned C and C++, which make you do all the work yourself. I had to release memory and do garbage cleanup manually, and I messed it up enough times as a teenager to realize I only wanted to use higher-level languages that managed it for me.

An *overflow* means that you have gone past the end of the amount of space allocated for your variable. You can overflow a buffer, string, integer, or the stack.

Let’s say you make a search field and set the variable to be five characters long. If the user inputs six characters, where does the extra character go? In a non-memory-safe language, it goes into the rest of the memory for your computer! On top of that, the rest of the memory on your computer is where the running program is stored. That’s right! All the instructions your computer follows to make your program work are also in memory. Figure 3.1 shows how this works.

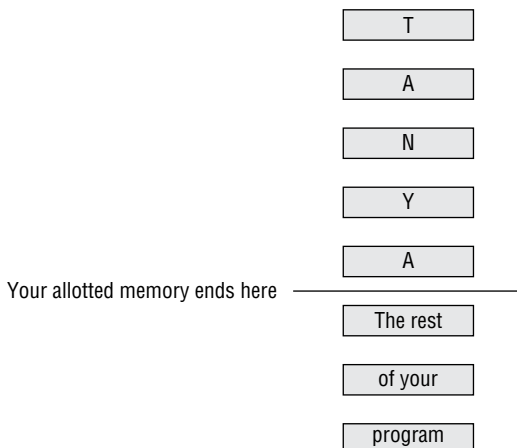


Figure 3.1: Properly Stored Variable

When your variable is overflowed, it goes into the rest of memory. If you have hit the part of your memory where the stack pointer is, you can then redirect the program to a new place. The stack pointer tells the computer where to look next for instructions. Your computer follows the stack pointer, executing the instructions in order until it is complete, or we enter a state of unpredictable behavior and/or memory corruption (the program crashes).

When someone performs a buffer overflow, they intentionally overwrite the stack pointer. They add their own instructions to the memory for the computer to follow, which is the code that performs the attack. The instructions from the attacker are called *shell code* and can be used to take over your web server. This means the attacker can execute instructions to the web server, and they will run with the same privileges that your application is granted on that same server (allowing them to access or modify information and other applications on the device). The attacker can do this from wherever they are in the world, remotely. When this situation happens, it is called remote code execution (RCE), and it is the most critical vulnerability possible for a web application. Look at Figure 3.2 to see how this works.

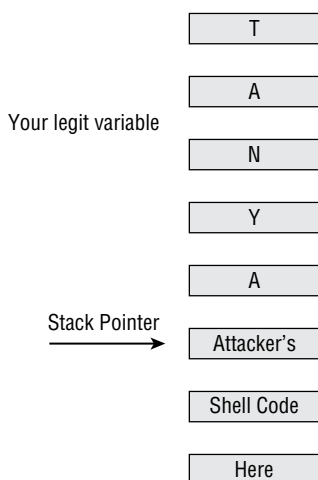


Figure 3.2: Improperly Stored Variable

How Do We Avoid Overflows?

Now that you know about overflow vulnerabilities, you will likely want to avoid them at all costs. The simplest way to do this is to avoid using programming languages and frameworks that are not memory-safe. For example, use Rust instead of C or C++.

What do you do if you don't have a choice? For example, there's a legacy application written in C or some other language that is not memory-safe, and

you must work on it. You don't have the time, money, or permission to rewrite it, so you just have to deal with it the way it is. Let's go over some best practices!

Avoiding Overflows

- Validate all your inputs. Verify your bounds for variables every time.
- Perform thorough testing for potential memory mismanagement.
- Perform code review, and prioritize reviewing for memory issues.
- Use a fuzzer to test every input of your application.
- Avoid using functions that don't perform buffer checks (for example, in C, instead of `gets()`, use `fgets()`).⁴⁴ See Table 3.1 for more items.⁴⁶
- Or use an automated tool or sanitizer that performs the buffer check for you.
- Use managed buffers and strings rather than raw arrays and pointers (see examples in Table 3.1 below, printed with permission from Snyk).⁴⁵
- Use a non-executable stack (i.e., data execution prevention [DEP]), which will mark the stack and, in some cases, other structures as areas where code cannot be executed. This means that an attacker can still inject exploit code onto the stack, but it will not run.⁴⁶
- Use an operating system with address space layout randomization (ASLR), which will drastically reduce the risk of your buffer being overflowed; it will not be able to overwrite the stack and take control of your program.⁴⁶
- Use canaries,⁴⁴ values inserted before the return addresses in the stack, and ensure that they cannot be overwritten. If they are overwritten or otherwise changed, the program stops.

Table 3.1: Safer Options

PREFER	AVOID
<code>std::string</code>	<code>char*</code>
<code>std::vector</code>	<code>int array[15]</code>
<code>std::cin</code>	<code>scanf</code>
<code><iostream></code>	<code><cstdio></code>
<code>Strncpy, strcpy_s</code>	<code>strcpy</code>
<code>Strncat, strcat_s</code>	<code>strcat</code>
<code>snprintf/sprint_s</code>	<code>printf/sprintf</code>
<code>fgets</code>	<code>gets</code>

OFF BY ONE

Bob took a programming class in high school and wrote a card game app in C. His program kept crashing, and he printed it out for his teacher, Mr. Ferguson. It turned out that he had made his counter start at 1 instead of 0 for 52 cards. In C, you must start at 0. It counts from 0 to 1, then 1 to 2, and so on. Whenever the counter got to the last card, it crashed. A few days later, the teacher returned the 10 pages of code to Bob, explaining his error. Bob felt very lucky to have a teacher who was so passionate about teaching programming that he could put up with Bob's 500 questions. Bob took all three levels of programming with his favorite teacher before graduating high school.

(De)Serialization

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file.⁴⁸ The reverse process is called *deserialization*.

—Microsoft Learn

Serialization allows you to save the state of your object and then re-create it perfectly after you've saved it or sent it somewhere. Your object might be extremely complex, and if you need to send it to another domain, using a web service, past a firewall, serializing it means it will not change during its journey. There are many situations where a programmer may choose to serialize an object. JSON and XML are the most commonly serialized objects and most likely what you will need to watch for, although serialization is still possible in .NET and other languages/frameworks.

Unfortunately, the deserialization process can be rather dangerous. When we deserialize an object, it is possible for it to contain malicious code, which then attacks the system we used to perform the deserialization. If an attacker can swap out the object you sent with something malicious, you are in for a world of hurt. This situation, accepting a serialized object from a source you do not control, creates a vulnerability called *insecure deserialization*.

As you may have guessed, the easiest way to avoid deserialization is to never serialize anything in the first place. That said, doing so is useful in many situations, so let's look at some of the ways you can reduce the risk.

If you must accept serialized objects, ensure that no attacker can ever swap it for their own. This means either encrypting or signing it before sending it, sending it wherever it needs to go, decrypting it or checking the signature, and then deserializing it. If you encrypt it with your own key, you should be able to trust that it has not been tampered with in any way. If you validate the signature, you should be certain you know where it came from and that its integrity is valid.

That said, if your object is sent from anywhere you do not control, if it is possible that someone could make changes to the object in transit, if it was provided

by the user, or if you do not have the option to encrypt or sign it yourself and then decrypt/verify after, you are in a situation where you must take significantly more precautions. This means the serialized object is from *an untrusted source*, which lands you in the insecure-deserialization situation.

For extensive details on how to defend your application while accepting serialized objects from untrusted sources, please see *Alice and Bob Learn Application Security*, Chapter 5, “Deserialization,”⁴⁸ and also the OWASP Insecure Deserialization Cheat Sheet.⁴⁹

Privacy (User/Citizen/Customer/Employee)

When creating software, there is generally a list of requirements given to you so you know what to build. However, some requirements are implied rather than stated explicitly, which can lead to confusion. Often, subjects like privacy and security requirements are implied rather than stated, and that means they don’t necessarily turn out the way your users/clients/customers may have envisioned. To avoid this, it is best to spell out exactly what you want, in as much detail as possible, in advance.

Privacy has come to the forefront of many technological conversations in recent years. Consumers and members of the public want to know what we (technology companies) are doing with their data. Although I covered this in *Alice and Bob Learn Application Security*, I think it merits additional word count.

Privacy: the state of being alone and not watched or interrupted by other people.

—*Oxford Dictionary*⁵⁰

Let’s expand on this definition. When we are online, we may be sitting alone in a room, but if our personal data is being stored and sold and our behavior is being tracked and/or interrupted by endless popups and notices about cookies, does that sound like privacy to you? It doesn’t to me.

Privacy is about self-determination and dignity. Not just data.

—*Abigail Dubiniecki*⁵¹

Privacy is also about the ability to make choices in private, such that you can avoid harassment, embarrassment, humiliation, and even violence. This point has never been more visible in our society, with large online hate groups using various forms of technology for harassing, doxing, swatting, and even killing those of us whom they do not find “acceptable.” When we respect the privacy of our users, we are also protecting their lives.

NOTE As a software developer, you may argue that when you program your software to meet the requirements of the client, and they require that you add functionality that violates users' privacy, you are "just doing your job." You are doing your job. But you are *also* being complicit in the violation of users' privacy. If you're reading this book, there's a good chance you care about security and likely also have some opinions about privacy. I'm not suggesting you quit your job and then burn the building down (please don't commit arson), but I am attempting to convince you that perhaps you could push back a little or explain how this might affect end users. Sometimes a gentle nudge or just saying something out loud in a meeting can entirely change the course of a project.

User, citizen, customer, and even employee data can be personal and/or private in nature. Personally Identifiable Information (PII) is data that can identify a user and make it easy for someone to steal their identity. Personal Health Information (PHI) are facts about someone's health, which could be used to increase their insurance premiums, block them from getting a job, embarrass them, blackmail them, or could even lead to death. Many types of information are fine on their own but, when combined, become an issue.

Another thing to pay attention to when it comes to privacy is that consumers care about it now more than ever before. Although the percentage of users who will abandon a product or service due to privacy issues is still small, it's growing every year. It is a strike against your product when compared to others and may not be worth the risk to reputation in the not-so-distant future.

TIP Many consumers do not fully understand the privacy implications of various products and their accompanying features. Because we are the creators of said products, it is our responsibility to consider the privacy implications to users' rights and protect those who do not understand what they are agreeing to.

SURPRISING SHARING

Back when Bob was in college, he and his three roommates often shared the same computer for internet access. Bob owned it, but the others didn't have computers, so they shared. One day a friend came over and looked through the cache while Bob was out of the room. He returned to have his friend ask him all sorts of uncomfortable questions about his internet usage. He told his friend it was not cool for him to look through the cache; it was Bob's business what he looked at on the internet! He showed his now ex-friend the door and then sat down to look through the rest of the cache. From sports sites to Nine Inch Nails, to Jewish Film Festivals, to all sorts of "adult content," Bob was very surprised by what he found. He set his cache to auto-delete from then on and put a password on the computer. No more sharing for Bob!

TIP Tip! Don't collect data you don't *require* for your system. It not only reduces your attack surface but also helps protect the privacy of your users. Win-win! If you are interested in learning more about privacy, check the field of Privacy Engineering,⁴¹ a new and important area of computer science and information management.

Errors

Some errors are also security incidents. Sometimes a combination of errors, or a certain type of error happening a specific number of times, or an error happening over and over again in a short period of time, can *become* a security incident. Being able to recognize what is and is not a security incident is something you will become better at over time with more experience. To start learning about it, read this section, and then work with your security team to learn threat modeling (more on this in Section 3 of the book), which will help you become very good at spotting potential security incidents.

NOTE When we talk about the “state” of an application, we mean a set of variables used to describe the behavior of the system at a particular time.⁵² A system is described as *stateful* if it is designed to remember preceding events or user interactions; the remembered information is called the *state* of the system.⁵³ An *unknown state* means the application is in a situation the programmer never accounted for when designing and coding it, meaning the application doesn't know what to do next. This type of situation tends to result in the application acting in unexpected ways, which sometimes leads to vulnerabilities or other security issues.

Every application should “catch” and then “handle” its errors. If we do not “catch” our errors, the application could crash, freeze, or fall into an unknown state, which could cause our application to perform unpredictable behavior. Security professionals never want to see an application fall into an unknown state unless we are performing security testing or vulnerability research because this is where many vulnerabilities are found. Catching an error ensures that you do not lose control of your program, leaving it to the underlying server software to handle your mistakes.

Once you catch an error, you then need to *handle* it. Handling means deciding what you want to do about the situation and then performing those actions. Usually it means throwing an error (generally a custom error message for that specific situation, never a system-generated message) to inform the user something has gone wrong, after which the application performs whatever action you deem necessary to remedy the problem. If the application was performing a transaction, generally you roll it back and start again. If it was trying to submit a form, generally the form is reset to its previous state. The action you decide to take when you handle an error is up to you and your client, and ideally it also follows whatever secure coding guideline or rules are set forth by your security team.

TIP Most of the rules for when errors become security issues are avoidable by having solid rate-limiting on every part of our applications. If we block the problems from happening, we won't need to sound an alarm.

Rules for when errors become security issues:⁵⁴

- Someone tries to log in more than 10 times in 10 seconds or more than 100 times in a minute. Humans can't type that fast. You may want to be more or less strict than this prescription.
- Someone calls your API incorrectly, over and over. How many times and how fast before it triggers a security alert or incident is up to you and your product owner.
- Someone repeatedly calls your API with bad formatting, but the calls change slightly each time. This could be a malicious actor enumerating your APIs (finding them all) or attempting to call your API manually as part of an attack or a precursor to an attack.
- The same IP address tries to make several new accounts for the same system in a very short amount of time.
- The same IP address tries to make more than one account with different email addresses, and none of the email addresses are verified.
- The same IP address attempts to change a password more than once, and the password reset link is never clicked. This is likely someone trying to break into someone else's account.
- A payment fails repeatedly. This could be someone who has had their credit card stolen or who is just plain down on their luck. Either way, there's a problem.
- Someone repeatedly attempts to browse to pages that don't exist. Several 400 HTTP errors in a row reeks of directory busting.
- Files are uploaded and rejected based on your precautions/verifications. If this happens for a few files in a row, something is wrong.
- Any transactions that you have to roll back are worth logging, but if you have to roll back something two or three times, this might be a security incident or attack of some kind.
- If you attempt to rotate or refresh your session ID (because it's been long enough and the client needs a new one), but the client is still trying to give you the old session ID, or if a session ID that you never issued is presented to your app, this is definitely a security issue.
- Repeated server errors, meaning a user has triggered it more than once or a few different ones in the 400 or 500 range. This might be nothing, but it could be something. HTTP server errors that happen more than once are something to investigate.

- Spikes in the CPU or RAM that just don't make sense. Especially if you have done extensive stress and/or performance testing in the past, and there are no external events that might explain the spikes (such as tickets going on sale for a popular band, election day, or the fact that you're a well-known journalist who has ticked off a group of hacktivists for the umpteenth time).
- Failures to authenticate to the database, an API, etc., that normally work fine. If your app suddenly can't authenticate to other systems, there's a problem.
- Anything anomalous that seems weird or unusual and unexpected is worth looking into. It might not be worth a security alert, but at the very least, it should be logged, and you should share what you find with your team.

This list is not exhaustive.

Errors, logging, and alerting should be organized as part of one system.⁵⁴ Ideally, there is one service for handling logs, errors, and alerts or three separate systems (one for each). One application should never have multiple systems for handling errors; they should all be handled the same way, ideally with the same functions. Code review and design review can help ensure that everyone follows the same plan and uses the same functions for a uniform approach to error handling. Life is much better for security and development teams alike when everything is uniform.

Logs should never contain sensitive information, such as PII, PHI, an entire user record, a complete session ID (the last four digits for troubleshooting are okay), etc. If the logs get into a malicious actor's hands, they should not be able to be used to further their attacks.

TIP Export your system exceptions by using an external error tracking tool such as Bugsnag or Rollbar to keep track of how many occurrences there are of exceptions and to monitor when new ones occur.

When creating logging for your applications, it's important that the logging part of the application is tested for something called *log injection*, a vulnerability where an attacker can create their own log entries or inject malicious content into the logs.⁵⁵

The last word on errors: every app should have a global exception handler. The global exception handler will catch any system error that your app created but that was not caught by any of your "catch" statements. It catches what the rest of your code missed. Theoretically, if you have done everything perfectly, it should never need to be invoked. That said, we have nets under trapeze artists for a reason! Your application should trigger some sort of alert if the global exception handler is ever called. It might not be a security issue, but it does mean you've missed something that needed to be caught, and you should investigate at your earliest convenience.

Logging, Monitoring, and Alerting

When Bob was a software developer, he recalls that every time he tried to add logging to one of his applications, he was told it was “too expensive.” Years later, he was part of a security incident for one of his apps. When the incident responders asked if they could see the logs, he responded, “We don’t have any. I was told it was too expensive.” They looked at him in shock. “What are we supposed to investigate? How can we possibly know what happened without logs?” Bob had no answer. When there are no logs, incident responders don’t know what happened, the source of the incident, or how to prevent it from happening again. It turns out logs are important to the security team, not only the software developers.

Web apps, APIs, databases, serverless apps, etc., all of them need logging. The software developers may have stuff that they wanted logged, but on top of that, the security team should ask for the following to be logged:

- Every security action/control. Someone logs in? Log it. Someone tries to log in and fails? Log it. Someone fails input validation ten times in a row? Log it! Then send an alert.
- All errors, system exceptions, and any other type of failure of the application.
- If the global error handler is engaged, log it and then trigger an alert.
- If possible, have your logs recorded in a way such that you can feed them into the SIEM (a security information and event management system, generally run by the network security team).
- When creating a log entry, include the following: a timestamp (UTC), the source IP address, what happened, whether it succeeded or was rolled back, system information or any error messages from the system, which part of the application it happened in (URL), identifying information for the user (username, email address, ID number), and anything else the security team asks for.
- Do not log PII, credit card numbers, passwords, SINS/SSNs, or any other sensitive information. Be careful to avoid any combination of multiple pieces of information, turning it into PII.
- Never log whole request bodies or object bodies. Use an allow list to only log what you need.
- Log every situation outlined earlier under the heading “Rules for when errors become security issues.” You want logging *and* alerting for each of those situations.

It is better to log more, rather than less, as long as you avoid logging sensitive information. If you are not sure, ask your security team. If they don’t know, ask

in the OWASP Foundation Slack channel. Joining the OWASP online community Slack platform is free, and it's full of experts who love to help others avoid security disasters.

If possible, use a standardized logging library across all of your applications for uniformity and to allow you to parse through your logs faster.

A note on log retention: if you delete your logs constantly, they may not be there when you need them. Security incidents don't tend to work within our preferred timelines, and the longer we keep the log information, the longer we can use it to investigate strange happenings. That said, I realize that storage costs money and logs are somewhat sensitive in nature. Try to balance cost, data retention, and the need to investigate as best you can.

Fail Closed

When computers have a problem with a transaction, it is important that we roll everything back and then start it again. When we do this, it is called *failing closed*, as opposed to *failing open*, which means allowing the problematic transaction to survive. When we fail open, there is a possibility that the transaction is incomplete, incorrect, a duplicate, or something else nefarious and/or unintended.

Although rolling back a transaction in your application and redoing the whole thing may seem like a bit of a pain, it's a lot less work than a potential security incident could be. Improperly or partially completed transactions can lead to all sorts of security vulnerabilities, data breaches, theft, race conditions, and worse. Always undo every part of the transaction and do all of them again, even if it "seems like it might be fine." Trust me, it's not worth the risk.

Locking Resources

My first full-time programming job was at a tiny company in Kanata, Ontario, Canada; it was the 1990s. I was 19 years old and couldn't believe I got paid \$12/hr to write code all day! And I got to listen to music while I did it! I wrote in a very low-level programming language, creating text-only backend systems, and there were four of us at the company, including the founder. I worked full time through the summers, and part time throughout the school year. My boss's name was Bill, and I thought he was the God of Programming. He *always* knew the answer. *It was awesome.*⁵⁶

One day I ran my program, and it updated something in the database, but when I checked after lunch, the record I had updated had changed *again*. But I wasn't the one who changed it. Predictably, I asked Bill if he knew what happened.

Bill: Did you lock the record when you updated it?

Me: Uhhhhh, no... What does that mean? Locks? What?

Then Bill smiled his knowing smile and taught me a new lesson!

He said “Imagine you save your document that you are working on to our shared folder. You have it open, and you’re updating it. But what if I open it too? And I start updating it, too? How does the computer know whose version to save? And what if you save yours and close it, but then after I save my version and close it? Whose version is saved to our shared drive?” As usual, Bill’s lesson had broken my brain.

If you have ever tried to open a file and it said it was in use, that’s because it was *locked*. Most databases do this by default nowadays; while you are performing a transaction, the record(s) is locked until your transaction is done, and then the file is unlocked (it becomes editable again).

Things have progressed quite a bit since the 1990s. Now most files lock themselves or allow real-time merging of changes. If you have used Google Docs before or during a meeting and seen changes happening before your eyes, this is real-time merging in action! You probably don’t need anything that fancy for your database files, but you do need to ensure that they are locked while you edit them. If you work in a low-level language or a non-modern database or have special circumstances, you may need to lock records, files, or other information manually in your code. Please be aware that although this is generally handled by modern database software (via transaction statements), that’s not always the case.

For further details on race conditions, the importance of locking records, and failing closed, please see Chapter 5 of *Alice and Bob Learn Application Security*, in the “Race Conditions” section and Chapter 8 of this book, in the section ‘Race Conditions.’

Enabling Password Managers

Password managers are software products that manage your passwords for you. By “manage,” I mean they encrypt passwords for you, save them, organize them, remember them, and even copy them into web forms for you. They can help you ensure that you have unique, complex, random, long passwords and even tell you if they are aware of your passwords being part of a data breach so that you can rotate (change) them immediately. Some will even act as an authenticator app, handling MFA challenges automatically for you. I’m a giant fan of password managers, as I believe they make life easier and help people stay safer on the internet.

When you are building an application with a login screen that has username and password fields, *please* allow users to copy and paste to those fields. Password managers need to be able to paste into the fields to function properly. In the past, there were misguided security professionals who preached against this, believing that allowing pasting into password fields would make it easier for malicious actors to brute-force those systems. In reality, it caused people to not be able to use extremely long and complex passwords because they were forced to type them out manually, which resulted in a lot of mistakes and

quite a bit of frustration for the end users. As a result, end users are likely to use shorter, less complex passwords for sites that disable pasting, resulting in easier-to-break passwords. This means their accounts are less secure. This is *not* a win for anyone.

To avoid this, all you need to do is *not* disable pasting into password fields. Please and thank you.

TIP Set realistic and reasonable password length and complexity policies. Don't set a password that only needs to be 8 characters; that's not long enough. Push for 12 or even 16 characters. On top of that, do not force strange complexity rules that are difficult for users to abide by. I'm sure you've been stuck at a password creation screen before, trying to figure out which one of the 12 things they've asked for you do not have. "Special character? Check. But not one of these four special characters? Check. Capital letter? Check? No repeating characters? Oh crap." Whenever possible, make it easy for your users to do the right thing.

Cryptographic Practices

If you work in a large enterprise or for a governmental organization, there's a good chance you have a large security team that has created a policy all about encryption, cryptography, and all sorts of other good security stuff. If so, you should follow the policies and their advice.

That said, if you do not work in a place where this is already sorted out for you, we have work to do!

First things first: *cryptography* is the mathematical field that is used to change data, using math, into a new form. Generally, we use cryptography to protect data and/or systems using it. *Encryption* is a two-way cryptographic method that is used to make your data unrecognizable and incomprehensible, to protect it; then you can undo the encryption (called *decryption*), and your data is exactly the same as before you encrypted it. *Hashing* is a one-way cryptographic method that changes the form of the data to make it permanently unrecognizable, with its original information being unrecoverable. You can never retrieve the original information from a hashed value (although some attacks can still happen against hashed values). Hashing should be used to store user passwords rather than encryption.

Generally, information security professionals recommend that we encrypt all data in transit (including the information going back and forth from the user's browser back to your web server) and at rest (saved to a database, a file server, etc.). For more sensitive systems and data, you can also encrypt data in memory and/or wipe memory immediately on completing whatever action you are performing. You should use the newest supported version of Transport Layer Security (TLS) with unbroken algorithms (no MD5, no SHA1, etc.; refer to the

NIST Cryptographic Standards and Guidelines for a complete list) and a long key (128, 192, or 256 bits for symmetric algorithms).

I wish I could tell you I was an encryption expert and knew all the answers, but I'm not. That said, I know who the experts are, and I can send you to them, confident that you will receive good advice for many years after the publication of this book. Here are my recommendations for current and trustworthy encryption advice:

- Cryptographic storage: Open Worldwide Application Security Project (OWASP) Cheat Sheet Series⁵⁷
- National Institute of Standards and Technology (NIST) Cryptographic Standards and Guidelines⁵⁸

THE FUTURE IS UNKNOWN

As I write this, we are still in a prequantum computing world. Once quantum computing is a reality, it may become possible to reverse hashed values. No one knows for sure that this will happen or, if it does, when it will happen. That's it. Send tweet.

TIP When information security professionals say “crypto,” they are referring to cryptography, not cryptocurrencies. If some dude bro in a bar tries to get you to “invest in crypto,” he means cryptocurrency. Unless you are into gambling, you might want to avoid that guy.

Strongly Typed Languages

We have previously covered the idea that if our application falls into an unknown state, the result may be one or more software vulnerabilities. This means that when we allow areas (such as variable declaration) of our application to be ambiguous, it can lead us into such a perilous state. If we use a programming language that is “weakly typed” and we do not declare our variables, or we declare them but then put whatever the heck we want into them (declaration type be damned), we have created *an area of ambiguity* in our program. And this can cause problems for us and our applications down the road.

Strongly Typed Languages

Strongly typed languages are ones where the type of variable is explicitly defined within the language; whenever you declare variables, they have to be one of those predefined types. You can't create your own types and then create variables using those new types; it's not allowed. If the language only provides you with string, char, int, bool, and float, that's all you get. Yes, I'm looking at you, C++!

Examples of strongly typed languages:

- C*
- C++
- C#
- GO
- Java
- Pascal
- Python
- TypeScript
- Rust⁵⁹

* C is strongly typed, but you are allowed to (try to) cast one var type to another. See the following tip for more info.

TIP In most strongly typed languages, you can't cast one type into another type, except for C. In C, you can "cast" an integer into a float. Well, you can *try* to cast it, but what happens next depends on your compiler's opinion of the rest of your code and the context of your cast statement. You may not want to try this at home, folks!

Weakly Typed Programming Languages

Weakly typed languages allow programmers to be more creative. You can create your own types in any format you can think up. This is exciting because the possibilities are endless! But from a security standpoint, it means we need to do more code review and testing around the invented types, because they could lead us to unknowingly create a vulnerability within our applications. Examples⁵⁹ of weakly typed programming languages:

- JavaScript
- Perl
- PHP
- Ruby
- Shell

It should be noted that not everyone agrees on the definition of strongly or weakly typed languages. Often, people use the terms *statically* or *dynamically* typed. Let's look at those, too!

Any programming language that is strongly typed can also be statically or dynamically typed. For static typing, the compiler checks the types as it compiles. This is great, because it can find any errors and tell you, and you can fix them before releasing your code out to the world. Dynamic typing is checked

during runtime (while your app is running!), and if the compiler finds something wrong, it will try to ignore it, but it might throw an error and your program may crash. That said, dynamic typing gives programmers more flexibility, which is an attractive option for many of us who make a living creating software.

You may be wondering at this point, what is Tanya trying to get at here? When you are choosing your programming language, keep in mind what each language offers so you can ensure that you double-check, test, or review your code, looking for any potential issues that apply. I don't advise that you do or do not use strongly typed languages, but I do want you to be very aware of the implications of the tools you use and the potential risks they may impose. Ensure that you test and/or perform code review for issues that tend to apply to however your chosen language may be typed. Also, resolve all related compiler errors. The stricter you are with yourself, the more secure your final system will be.

Domain-Driven Development

Domain-driven development (DDD) means designing your objects (in object-oriented programming [OOP]) around real business actions/needs/activities/etc. As an example, We Hack Purple sold online, on-demand courses. If we were designing an academy platform to host those courses, there would be “course,” “student,” and “professor” objects. Those objects would mirror the real-life business model; courses would have students who were enrolled, their grades (pass/fail), whether they had completed the course yet, and so on.

When we design our objects based on the real-life business model, designing IAM becomes very intuitive. It's obvious that a “student” object shouldn't get to decide their own grade or the grades of other “student” objects. This knowledge of the business domain will inform sound decisions around access, permissions, etc., as you build your app. This is one way that DDD is helpful for security!

When designing using DDD, there are several components to be aware of: entities (objects that you can classify the data inside of), value objects (sort of like a constant that you can compare against), data events (events that are important, tracked, and logged), aggregates (a bunch of domain objects that are linked together: for instance, a student object can have aggregates for every course they've completed), and repositories (grants authorization).⁶⁰

An excellent example of this was given in a talk by Michael Koppmann, “Typed Security Preventing Vulnerabilities By Design,” at the Sec4Dev conference in Vienna, Austria, 2022.⁶¹ When we accept an email address from a user, generally we want to run it through some sort of validation process (to ensure that it's a genuine email address and that it's not malicious). However, when we run it through the validation process, normally we don't mark the email field or object in any way to make it clear that it has or has not been validated. Thus, a programmer who didn't know better could think that the email address variable

had already been validated when it hadn't been and then use it to make business decisions, store it in the database, or reflect it to the screen, unknowingly creating a vulnerability. With DDD, we can create an "email address" object *and* a "validated email address" object. We accept the original input from the user into the email address object and then run validation on it, and if it passes, it is saved into the validated email address object. It is very easy to write or review code when it is this clear: obviously, you should only use the validated object for making business decisions. This lets you see immediately if unvalidated user input is being used in error. If you designed all user-input objects in this way, you could eliminate the possibility of many vulnerabilities, such as Cross-Site Scripting (XSS) and SQL injection (SQLi). *Win!*

The key thing with DDD is that you can design security into the model from the very beginning and it is an intuitive part of the design process. It's not tacked on later; it's naturally part of the design progression. Anything that can help guide you or others down the more secure path is a win in my books, and if domain-driven design and development sounds attractive to you and your project team, I encourage you to give it a try!

TIP When discussing DDD, sooner or later, pretty much everyone refers to Eric Evans' book *Domain Driven Design*.⁶² Read it to learn way more on this topic!

Memory-Safe Languages

We have already discussed overflow vulnerabilities in this chapter, why we want to avoid them whenever possible, and that the easiest way to avoid them is to use a memory-safe language. That said, I wanted to talk about this a little more.

When we choose a memory-safe language, it may mean a rewrite of an older application, it may mean we need to send a bunch of employees for training, and it may mean a bit of a learning curve for a project, slowing things down. But we also avoid every single kind of memory security vulnerability, all of which are considered critical in the risks they pose. When we choose a secure default by choosing one language or framework over another, we can eliminate several types of potential mistakes that could cause serious security issues down the road.

RANT

If you are reading this book, you are likely a subject matter expert at work (also known as a SME, pronounced "sm-eee," if you feel the need to say it out loud). This means you have the ability to influence others at work, for better or worse. Even if you are just an intern, you can point out that "Maybe it's worth the extra effort to learn Rust,

just so we can avoid all those pesky overflow bugs.” In most cultures, staying silent implies acceptance or approval of whatever has been said, decided, or announced in your presence. It is my hope that after reading this book, you will feel more confident to speak up and suggest small (or maybe even large) changes to the way things are done in your workplace. One of those things that I hope you will feel confident talking about is choosing a memory-safe option whenever it’s possible to do so.

Chapter Exercises

You can find the answers to these questions in Appendix B, “Answer Keys.”

1. Is there ever an appropriate reason to use a language that is not memory-safe? Yes or no?
2. Applications written in strongly typed programming languages are always more secure than weakly typed languages. True or false?
3. Data should be encrypted in transit, at rest, or both.
4. Blocking the ability of your users to paste into the password field makes it more difficult for users to use password managers and makes it statistically more likely that your users will use shorter, less complex passwords. True or false?
5. Privacy includes not being interrupted when you are alone. It means not being tracked across the internet. It means not having your private data sold without your knowledge. Privacy, as defined here, is currently being violated by many websites and web applications across the internet every single day. True or false?
6. Databases need to be protected from several different angles, not just from database injection vulnerabilities, such as SQL injection. True or false?
7. We don’t need to label our data after we classify it because everyone should know what classification it is just by looking. Labeling it is redundant and not necessary. True or false?
8. Source code is intellectual property, and you should take precautions to protect it. True or false?
9. Deserialization can be avoided by never accepting untrusted serialized objects. True or false?
10. It’s okay to use production data in a testing environment because “we trust our employees.” True or false?

CHAPTER 4 Achieving

Now that you have improved your secure coding knowledge in the previous chapters, you are ready to *achieve*. This next chapter of agnostic secure coding advice is more advanced but still very doable.

This chapter will cover more than just coding; we will delve into secure design and the concepts (such as creating secure defaults, writing reusable and readable code, and guaranteeing that important decisions happen on trusted systems) we will apply to ensure that the final product is rugged, tough, and secure.

We will discuss how to manage our dependencies, avoid brute-force attacks, and perform system hardening and hostname validation. We will cover the merits of allowlists versus blocklists, paying back technical debt, and so much more.

This chapter will focus on the larger decisions that we make to allow us to create maintainable systems that remain secure over the long term or that slowly lead us to drift into failure.⁶³ The entire chapter is biased toward investing time, effort, and budget now in efforts to avoid potential catastrophe later.

If you speak to anyone who has worked in security at an organization that has had a large data breach, most of them will tell you that they knew about the issues for months or even years before the breach happened. They had been fighting for the budget to repay their technical debt or add integrity checks but had been denied, often repeatedly.

This is where you, as the one creating and maintaining the software, can be of great assistance to your organization's security team. When these decisions are being made, you can interject with relevant information to help the entire team

make the best choices for your organization for both the short and long term. Whether you participate in creating an option analysis to decide which technology to use for a project, insist on adding your system development life cycle (SDLC) security activities into all project schedules, or set aside time each month for chipping away at your team's technical debt, you are fighting the good fight.

I hope this chapter helps you steer your entire organization toward a more secure future.

Secure Design

Designing systems securely means planning out systems with the intention of making them dependable, reliable, and safe for both our users and our data.

Quite often, when we hear security researchers and penetration testers talk about business logic flaws or security flaws, they mean security flaws *in the design*. A security *bug* is when we've coded something incorrectly, and it created a vulnerability. But when there is a *flaw*, that means the problem is in the way the system was *designed*. If you continue listening to security researchers and penetration testers, at some point they are likely to complain that automated tools are "unequivocally terrible" at finding business logic flaws, and they are absolutely right. This is why they all have jobs: because humans are much better at seeing these sorts of things than computers are.

GRC EFFICIENCY

Bob used to work at a large government department with a giant GRC team (Governance, Risk, and Compliance). He recalled them making him and his team jump through many hoops to be allowed to release their systems. One day, a member of the GRC team presented him with an 80-page document; it was an architecture template they wanted him to fill out. He showed it to his tech lead, who had the same reaction. How would they ever be able to complete this on top of all the other work they already had to do as part of this project? Bob was told it was mandatory, so a member of his team was dedicated to completing the document. After the project was over, Bob worked with the GRC team to shorten the document, speed up the process, and automate some of the answers. GRC is important, but we also want to be as efficient as possible with their project time.

THE SOCIAL BUTTERFLIES OF SECURITY

Last year, Alice worked at a company that was acquired by a larger company. The GRC team from the parent company worked hard with her entire IT department to ensure that their products and processes were documented the same way as at the parent company. At first it seemed like a lot of work, but after about 90 days, she started to see why: they figured out how to reuse code, provide services to each other, work together better, and make sure they met all the audit requirements. Every time

someone on her team had a question, GRC knew either the answer or who to talk to. When they went through the annual audit later that year, which had always caused Alice giant headaches, they passed with flying colors and little effort. It was amazing! She'd never thought of GRC as being social butterflies before, but this team was the "glue" that held together the whole IT department.⁹¹

One of the ways humans can spot and prevent design flaws is by performing *threat modeling*, which is a process for identifying potential threats to your system(s), documenting them, and then (hopefully!) finding a way to mitigate the more disconcerting issues you have discovered. Threat modeling can be anything from a very formal process, with lots of documentation, to just a brainstorming session with some notes afterward. We will cover how to start with threat modeling in the third part of this book.

Another way we can try to prevent security flaws is to perform an architecture and/or *design review*. This process also varies greatly from organization to organization, but generally, some sort of diagram is drawn, questions are asked, discussions are had, documentation is created, and at the end, there is likely to be some advice on how to change or add things to our design to make it significantly safer. This process can also be extremely formal and time-consuming or faster with less documentation.

When designing systems, we can look to well-known *design paradigms* that are known to be secure, reliable, and resilient. When we follow a paradigm, we have a process that helps ensure we are organized and thoughtful in our task (as opposed to building software in an ad hoc way, which tends to be unmaintainable and only works for simple systems). Examples of design paradigms are data-driven, object-oriented programming (OOP), procedural, functional, domain-driven design,⁶⁴ and so on.

SNEAKY COLLEAGUES

Bob was struggling with the GRC team's demands. They were always asking him to create giant architecture documents, but he could barely keep up with his current tasks, much less add more. One day he ran into his colleague, Heidi, and asked her how she managed to keep her projects on time *and* get all those documents to the security team. Heidi confided in Bob that she copied the templates, added the names of her projects, and then put them into their OneDrive with strict permissions so that no one could see them. She would email the link to the GRC team and wait to see if they requested access. She said that most of the time, no one did, and she saved the time she would have spent writing it to work on her projects. If someone did request access, she would drop everything and fill out the form. She said that in all the years she had been doing it, she only had to fill out the forms twice. That's hundreds of saved hours. . . . Bob didn't like the idea of lying to his colleagues; he felt that was wrong. But he also didn't like the idea that his documents might not be read. He decided to talk to the GRC team about it.

Secure design concepts can also be applied and built into our systems by making them a part of the design itself. Examples of secure design concepts include

- Assume breach
- Least privilege
- Zero trust
- Defense in depth
- Attack surface reduction
- Usable security
- Security by obscurity (least effective, used sparingly)
- Supply chain security

If you design your app to have the smallest possible attack surface, there will be less to attack. If you ensure that your application's security features are easy to use, you will have very few users who attempt to go around them. When we build the security features into the way the application works, it makes it very difficult for attackers to exploit our systems.

How much is “enough” (design) security?

It's important that we decide how much effort and money to spend on security based on several factors (listed in a moment). Just as we do not want to leave our systems defenseless and weak, we also do not want to “overdo” security by spending so much money, time, and resources that the end product is too expensive and we are unable to sell it. We must generally strike a balance between security and profit. That said, when human lives, national security, or other special factors are in play (things that have a value that cannot be measured with currency), we will often choose to do “more” security based on non-monetary factors.

Deciding “how much” security is enough:

- How much risk could this system pose to our organization?
- Is the data sensitive? How sensitive is it? Does this data pose a risk to our users?
- Does our organization face advanced threats (nation-states, hacktivists, direct business competition, etc.)? Or are we just facing the “regular” threats on the internet (script kiddies, criminals, and users who make mistakes)?
- Do we have enough trained people to do this workload? If not, do we have budget to hire more people? Are there people available to hire who are trained and ready to do this work?

- Do we have enough time to support this new process?
- Is doing this process more important or valuable to our organization than another? We want to prioritize the security activities we choose based on how much value they add versus the cost in money, time, and resources they consume. If costs outweigh the value, we may need to implement more balanced security activities.

Although it is often the case for security professionals that we always feel there is more work we could do, eventually we have to draw the line. If it's possible to do three very thorough threat models on our most important system versus doing a faster threat-modeling process for all of our systems, which option is better? You will need to discuss this with management and choose which one you feel will reduce your organization's risk the most.

ALICE'S FIRST TASTE OF THREAT MODELING

Alice was invited to a threat-modeling session by the new person on the security team, Faythe, who told her that all new projects would go through this process. The session included someone from the AppSec team, Alice (product owner), and someone from the technical team. They followed Adam Shostack's 4 Question Framework for Threat Modeling.⁶⁵ It was surprisingly unpainful! Alice had no idea it was possible to uncover problems so quickly, and they even came up with some solutions. Alice felt a lot more confident in this system compared to the old one, where she filled out endless paperwork and never received helpful feedback. For the first time, Alice felt that the security activities showed a clear return on investment. She was used to the security team making them fill out paperwork, as opposed to talking to them; this new process felt like a breath of fresh air.

TIP Try starting with the 4 Question Framework for Threat Modeling⁶⁵ by Adam Shostack. Asking these questions during a threat modeling session can help you find threats in no time!

- What are we working on?
- What can go wrong?
- What are we going to do about it?
- Did we do a good job?

Dependency Management and Supply Chain Security

The supply chain for a product is all the things you need to build it; this includes the tools and the materials you use. For a can of soup, you would need all the ingredients that go into the soup, all the things you need to prepare the soup

(a machine for canning, a large pot, and a stove, perhaps?), as well as the packaging (the can and its label). The supply chain to make software means the programming language and framework, dependencies (such as libraries, NuGet packages, Ruby gems, npm packages, and any other “includes” or third-party components), continuous integration and continuous delivery / deployment (CI/CD) system (if you use one to release your code), the operating system you host the app on, the IDE to write your code in (even the extensions you install into your IDE that help you write your code), your version-control repository, and anything else you need to build your software and maintain it.

Let’s separate dependencies from the rest of the supply chain and start there.

Dependency Security

My friend Rana Khalil⁶⁶ explains it like this: imagine that you are having a party, and you invite 10 friends. Those friends each invite 10 more people. Then *those* people invite 10 *more* people, and suddenly you have a house party that is completely out of control, and your house is full of people you’ve never met before. Dependencies in software can sometimes be like this: you include five dependencies, and each one has *its own* dependencies (which are called transitive dependencies—anything in the chain past the ones you included on purpose), and so on. Your app that you thought had only 5 libraries actually has 72 libraries once the building of your app is complete. All that code, whether you wrote it or not, is part of your app and therefore adds potential risk for your organization.

TIP In *Alice and Bob Learn Application Security*, the first book in the *Alice and Bob Learn* series, we covered what a supply chain is in detail. Read that to learn even more on this topic!

HOW Log4J RUINED TANYA’S CHRISTMAS

In December 2021, a perfectly innocent logging library in Java called *Log4j* ruined the Christmas vacation plans of thousands of incident responders when a critical vulnerability was found in it that was easily exploitable. Nicknamed #Log4Shell, this vulnerability wreaked havoc across the internet, with attackers only needing to copy and paste a string of characters into the address bar of a browser to test if the application was vulnerable. The open-source maintainers worked extremely hard to put out fixes as soon as possible, but in the meantime, many organizations were exploited, costing untold amounts of overtime, heartache, and breached data. The problem is that our entire industry here is not the open-source maintainers; our entire industry depends on unpaid work from (wonderful) volunteers, who do this work *as a hobby*. We are building professional systems on top of hobbyists’ work. Our industry needs to find a way to support this work (financially and with professional resources), especially to ensure that the libraries we use every day are secure and safe to use.

Checking If Dependencies Are Safe to Use

Over the past decade, the application security industry has started to release automated tools to help figure out which dependencies are safe to use and which are not. This type of tool is usually called a *software composition analysis* (SCA) tool but is sometimes called any variation of the term *software supply chain security tool*. Most of them have a list of known bad dependencies (ones with vulnerabilities that are public knowledge or that the vendor has discovered itself via research), and they compare the components you have included in your app to see if you have any known-to-be-vulnerable library versions.

Most of the tools currently on the market just tell you if you do or do not have a vulnerable library in your app. Newer SCA tools are starting to report whether your app is calling the function inside the libraries that have the vulnerability in them; this is called *reachability*. Generally, if you are not calling the vulnerable functions of the library, it does not create an exploitable vulnerability in your code. Although this means you will likely want to upgrade off of this vulnerable version at some point, it also means you don't need to drop everything and run out and fix it that very moment. If you are not calling the function with the vulnerability in it (the vulnerability is not "reachable") from your code, it is usually not exploitable. These tools generally present some sort of upgrade path so that you know which version of the library to switch to, and some of them even create a pull request (PR) for the change on your behalf, which can be rather helpful. As these tools mature, it's likely that at some point they will not only be able to tell you if the vulnerability in the library is exploitable or not but also, hopefully, even be able to fix the vulnerabilities for you.

Supply Chain Security

There's more than just dependency management involved in the software supply chain. Everything you do or use as part of the process to create the software is part of the supply chain, and all of it needs to be secured. This includes:

- The system you write your code on, whether it is in a managed sandbox or on your laptop, needs to be secured against viruses, malware, and unauthorized access with (at least) regular backups.
- The network and infrastructure your company uses to host software you need when you deploy. You must secure it before you deploy it and continue to ensure that it is secure after your app is out in production. This goes for every bullet point on this list.
- Any communication (Slack, Microsoft Teams, etc.) or bug tracking systems (Jira, Bugzilla, etc.) you use.
- Infrastructure you use to build your software, such as the development server or a sandbox area where you play around with new technologies.

- Your IDE (the place you write your code), all your tools that are plugged into it, and the way you have it set up.
- The place you save your code (often called a “repo,” version control, code repository, or source control) needs to be backed up and protected from malware and unapproved user access. Ideally, you will have multiple backups that are geographically distributed, as this intellectual property is extremely valuable to your organization. If you use a cloud provider or SaaS (Software as a Service) solution, you may want to inquire about its backup policies and processes.
- The location (package manager or website) where you get your third-party components (libraries) and/or containers from must be a trusted source.
- The path the libraries and/or containers take to get into your build environment must be secure (encrypted in transit). You can do an integrity check (validating the checksum or signature) to verify that what you received is genuine.
- Your supply chain may include hardware, especially if you create IoT (Internet of Things) products.
- All the configurations that you use in the development process are also part of the supply chain. Think of how long it took you to set up your development environment; that’s configuration time!
- Your CI/CD system needs to be secure from people changing settings who should not have that type of access (for instance, disabling security tests) and backed up somewhere safe (if you lost your CI/CD, it would cost time and money to set up and configure again).
- Any third-party APIs that your systems call need to be called securely (encrypted in transit) and need to be trustworthy in themselves (you should be performing authentication and authorization when you connect to them and validate any data you receive from them before using the data).
- Any third-party data your system depends on needs to be encrypted in transit and follow the same steps as for APIs (AuthN and AuthZ, plus validating data before use).
- Sign the code your team created after it’s built but before it’s sent to customers. This helps your users know that they are using the right version.
- Throughout this process, you will also likely perform one or more forms of static and dynamic analysis of your applications to ensure that your team is not releasing insecure software to users. Vulnerabilities found during these tests that create unacceptable organizational risk should be fixed and retested before release.
- The build artifacts (the built code, reports, logs, etc.) should be stored somewhere safe and backed up.

- Verifying that your licensing is in order should also happen during this process. Although licensing is technically not a cyber security risk, it is still a legal risk to your business and thus needs to be resolved if an issue is found.
- Verifying that your app follows your organization's privacy policies should also happen at some point in this process. This check may be manual in nature.
- If you are releasing a brand-new app, API, or other software system, you should add it to your organization's inventory of software assets. If it's replacing an old version, perhaps with a complete refactor, or your API version goes from 1.338345 to version 2.0, ensure that the original version is properly decommissioned. You may also want to create a software bill of materials (SBOM) at this time.
- Once the applications are in production, they should be monitored, they should perform logging, and these logs should be fed into some sort of monitoring system (such as a SIEM (security information and event management) system).

You may see software supply chain attacks in the wild where a threat actor (perhaps a nation-state or organized crime group) creates a vulnerability intentionally in a library, publishes it out into the open-source ecosystem, and then, when their target (the victim organization) downloads it, pounces. But this is not a common occurrence; these types of attacks are rare and are usually launched to find their way into a very secure organization (such as a government department or private company that holds something of extraordinary value). More commonly, with supply chain attacks, an unintentional vulnerability is discovered in an open-source library, and malicious actors then try it out on as many organizations as possible. These types of attacks are not often targeted against only one specific group or organization, as they are expensive, complicated, time-consuming, and dangerous to carry out.

The targeted supply chain attacks that we hear about in conference talks, blogs, and podcasts are usually very serious, unique, against a high-value target, and/or ingenious in nature. But smaller attacks happen all the time, and any team that lost their data, code, or CI configurations can tell you that a successful supply chain attack can lead to devastating consequences. Don't let this happen to you or your team: secure your systems as you create them, and make the security steps a part of the processes you always follow.

WARNING When you are part of a merger or acquisition, malicious actors often see this as an opportunity to strike. Attacking smaller or less secure entities in an effort to get into larger organizations has worked many times in the past. Be extra diligent with your security procedures when combining two organizations into one.

PROTECT YOUR REPO

Alice overheard two programmers waiting for the elevator talking about “an unmitigated disaster” and couldn’t help herself—she started eavesdropping! They explained how the manager had given the new co-op student (intern) read/write access to the code repository, and the student had deleted *everything*! Their manager was in the process of trying to recover from backup and had sent the team to “go get some coffee or something” while he had a private meltdown in his office. Oh my!

Secure Defaults

What is “secure by default”?

Secure by Default products are those that are secure to use out of the box, with little to no configuration changes, and are available at no additional cost, such as multi-factor authentication (MFA), gather and log evidence of potential intrusions, and control access to sensitive information. – Cyber Security and Infrastructure Security Agency (CISA)

When we are building software, there are two options for “secure by default”: (1) we build secure defaults (settings that are set to the most secure settings) into the products we create for our users, or (2) we set up our development environments such that the development choices we are offered and that are the “easiest” to choose, are also *the most secure* options. This way, both the developers and the end users are more likely to take the more secure path more often. This is also often referred to as a “paved road.”⁶⁷

Let’s start with our users!

Secure Defaults for Users

As designers of software, one of our responsibilities is to protect our users from threats, and sometimes those threats are themselves. Not all users of our software will be technical, patient, or mindful of the way they use software at all times. Even I can be seen from time to time mindlessly clicking “agree” on a terms of service statement because I’m in a hurry, I’m feeling impatient, or I assume “it will be fine.” Users often do not change settings on their own unless they need to. With this in mind, if we choose default user settings that are secure, it is unlikely that users will take the time to change them. I realize that this is somewhat manipulative, but I feel justified in this one instance; this is the best opportunity for us to protect our less technical and/or cyber-aware users.

What are some examples of secure settings we could set up for our users?

- Offering multifactor authentication and, for important systems, making it mandatory

- Verifying that the passwords they choose are not passwords + email combos that have been in a breach or that are incredibly common (such as “Password1!” or “Winter2025”)
- Creating backup codes or an emergency kit for if they lose their passwords or their account is otherwise compromised
- Telling users when the last time they logged in was and whether they have attempted to change their password
- Telling users which devices are logged in and allowing them to log them all out at once
- Telling them if they have approved third-party apps to integrate with your app and reminding them to verify that they still want those approvals in place
- Remind users if they have other users who have access to their account
- Have backup user accounts, such as another email address owned by the same person, in case they are locked out of their account
- Creating password policies that work well for users, especially ones that allow for or encourage password manager usage: do not ask users to change their passwords every 90 days, only ask them to change their password if there’s a suspected breach, allow copy and pasting from password fields, and do not ask for incredibly complex password rules (which generally annoy users and rarely increases security)
- Ensuring that sharing settings (data, images, location, etc.) default to minimum, and setting user data to private by default (e.g., for social media apps)
- Ensuring that systems do not broadcast by default unless that is the system’s main purpose (for instance, a WiFi router at a café needs to broadcast)
- When providing access to other users, applying least privilege as the first option
- If encrypted and non-encrypted channels must be offered, defaulting to encrypted

Hopefully, you get the picture. Whenever possible, set up your users to make more secure choices by making those the default options.

NOTE Although you (the software developer reading this) might think you’ve found a *workaround* for a problem, if you asked a security person, they will likely tell you that what you have found is called an *exploit*. Software developers perform “hacking” quite often; we just call it by a different name: troubleshooting.

Secure Defaults for Developers

Now that we have tackled the easy part, the regular users, let's talk about the most powerful software users around: software developers.

Software developers are like water: they can effortlessly go around whatever you attempt to block them with.

Said by me, over and over, in client meetings, when explaining that a technical safeguard won't stop their developers from doing whatever they want.⁶⁸

We want software developers to be able to make more secure choices, easily, while still being productive. In an ideal world, we also want them to enjoy their work! This means we need to find ways to help them make more secure software while creating the least amount of friction for them. One of the ways we can do this is via secure defaults (aka paved roads). Examples of paved roads include the following:⁶⁹

- *Secret management*: Creating policies for how to store secrets, implementing a secret management tool, providing documentation and training on how to use the secret management tool, and then creating a precommit hook for when developers check in their code that rejects code commits that contain anything that looks like it might be a secret. If you can add a link to the documentation on how they *should have* handled the secret, all the better.
- *Security policy compliance*: This could mean having your cloud security posture management (CSPM) system make a prioritized list of instances of broken policies, ordered from “fix this right now” to “get to this when you have time.” Perhaps someone has administrative rights on a Platform as a Service (PaaS) that hosts their app, but they do not have MFA on their network account. That's a “fix this right now” sort of thing.
- *Using static application security testing (SAST)* when checking in code to block any known antipatterns (bad functions, previously identified bugs, code that doesn't abide by your coding standard, etc.).
- *Helpful documentation*: Whenever possible, create a one-pager with specific, concise, and actionable security advice for every type of technology, integration, or tool usage. Distribute and socialize among your developers.
- *Self service*: Because software developers are power users, they are not used to waiting on others for technical tasks. Instead of wasting their valuable time with waiting, give them access to do things for themselves. Let them run their own scans, submit their code for review, approve each other's code, validate results, access golden images, etc. Any time you

can avoid them waiting on the security team and let them do the work for you is a win for everyone.

- *IDE tooling*: Whenever possible, select AppSec tools (SAST—dynamic application security testing (DAST)—software composition analysis (SCA), linters, etc.) that also have IDE plugins, allowing software developers to take control of their own destinies by testing and evaluating their code before they even check it into source control.
- *Automated checks* can be added for when you check in your code, in your IDE, and in the CI/CD. You can also add automated scans to the code repository weekly to see if anything new can be found, even if old code hasn't changed. Sometimes there's a new library vulnerability that you should be aware of, so you can dust off that old code and upgrade off of it.
- *Unifying your tech stack*: Although trying to dictate which programming languages, frameworks, code repositories, bug trackers, and other tools the developers use is likely an uphill battle, the more you can get them to standardize, the easier it is to secure (and keep up to date). If 100% of your code is in the same repository/version control system, the security team only needs to scan one repo for complete coverage and can easily find a tool to do it. Trying to scan seven different kinds of version control systems in 21 different places makes the security team's job infinitely more difficult and may result in them needing to purchase multiple tools, meaning they have less to spend on other security measures.
- *Offer training* to everyone on how to do their jobs securely. This means how to use the security tools you give them, what the processes are, and what to do if there's a security incident. This way, they know what to do, and over time they may even invent their own (secure) paved roads.

Another item you could add here is asking your developers what they wish could be automated or self-service and then figuring out if you can do it. While you are at it, examine your own processes once a year (or more) to see if there is anything you can do to speed them up or make them simpler or more efficient.

Readable and Auditable Code

Have you ever had to fix a bug in someone else's code (or your own code) and it was an utter disaster? You probably spent hours just trying to figure out what the code was doing before you could even start to work on the bug. This is why we need to create readable and auditable code for whoever has to work on our code *next*. And believe it or not, after enough time has passed, that person who has trouble understanding the code might be you!

Let's start with what readable code means: others can look at your code and easily understand what you were trying to do. If someone can't understand

what the code is supposed to do, how can they possibly make changes to it safely? If they don't understand the purpose of a function, imagine just how badly they could break it.

Code auditability is different. This means the code can easily be reviewed for security and compliance purposes. It's not quite the same thing as readability. Ideally, you have put in comments to explain "This is a security control that does X" or something of the sort, so the auditor knows what you are trying to do. If someone can read your code and comments and see clearly where each security control is and what it's supposed to do, and can read it to know that it's implemented correctly, you have succeeded in making your code easily auditable.

BOB FOUND SURPRISING SECRETS

Bob was asked to review an application for security before his company acquired the company that made it. He was given only a few days, so he got to work fast! He ran a SAST, a DAST, and an SCA tool on it and found the most interesting results with the static analysis tool: the live banking credentials for the company! *Holy smokes!* Bob immediately called the creator of the application and asked them to rotate their credentials (create new passwords) and then helped them switch to using a secret management tool instead of putting the secrets directly into the code. Bob really earned his paycheck that day, both for finding the important security issue and with his integrity by reporting it immediately and fixing the issue.

Now that we understand what we're trying to achieve, where do we start? Let's go!

- Follow the naming conventions of your organization's coding standards. If you do not have any, perhaps you could suggest that your team make some. If everyone follows the same naming conventions for variables, functions, objects, etc., your code becomes instantaneously easier to understand.
- Add helpful comments, especially for security features (protecting the user) and security controls (protecting the system). Often, programmers add a comment at the start of a function to explain its purpose and anything else important for the next coder.
- Spacing and formatting. Drilled into us at any coding course or bootcamp is that using tabs to denote a loop, spacing between functions, and using white space to make things easier to read is one of the ways to make our code more pleasurable to work on. You can use a unified automated linter to help with this!
- Keep it simple. Whenever possible, avoid complexity in your programming. Although sometimes it might be fun to show off our amazing skills

to others, this is not the place to do it. Making things overly complicated means other programmers will have to spend more time, and thus also more money, maintaining the apps that you build. On top of this, making things overly complicated can introduce unintended vulnerabilities, which is why some people say that “complexity is the enemy of security.”

- **Peer review.** Ask a colleague to look over your code for quality and security issues. This could be an official and formal part of your SDLC or just you asking a buddy to review it for you if you feel something needs a second set of eyes. Asking a junior person on your team could help you find out very quickly if your code is easy to read or not. If possible, make this a formal part of your SDLC!
- **Take matters into your own hands by auditing it yourself with tools!** Get a license (hopefully from your security team) for whatever tool(s) they use to scan your code, and then run them on your code. SAST and SCA tools aren’t that difficult to use (usually), and the security team would likely be happy to teach you how. You can see the results for yourself, fix them, and *then* submit your code. Win-win!
- **Paying back technical debt.** One of the things an auditor will look for is technical debt with security implications, which usually takes the form of out-of-date (several versions behind the current version) or known-to-be-vulnerable libraries and third-party components. If you can keep your technical debt small and repay it often, you are less likely to have an auditor find problems in this area.
- **Obey your licensing agreements!** When adding third-party components, ensure that you read the licensing agreement and that your use of it will be compliant. Although this is a legal risk, not a security risk, any skilled auditor will point out licensing problems in their final report.⁷⁰
- **Legislation and compliance:** someone on your team or in your organization should verify that your system’s requirements and design abide by any laws or compliance regulations that you need to follow (such as PCI compliance, California Privacy laws, or GDPR).⁷⁰
- **Refactoring:** sometimes it makes sense to rewrite or refactor your code. If you do this, be sure you follow the nine steps before this one to ensure that it’s readable and auditable! Refactoring tends to produce higher-quality code, and with this list and some determination, you can make your code a thing of beauty.
- **If possible, add diagrams to your documentation to help people understand it better.** An illustration of your design, data flows, architecture, or anything else you think might be helpful can go a long way in making yourself understood.

TIP If you don't make time for technical debt, it will make time for you! Many software-related security incidents you face will be more difficult to respond to if you have a large amount of technical debt. When you have a fast release process, low tech debt, and the ability to release virtual patches (more on that later), your team can be a lean, mean, incident-responding team.

It's normal to want to "just get your code working," but remember that taking the time to write maintainable, readable, and auditable code will pay off in the long run. Your code will be easier to maintain, more secure, and less prone to errors. Also, your colleagues will like working with you more!

Important Functions Happen on Trusted Systems

ALICE LEARNS A POWERFUL LESSON

Alice remembers going to something called a CTF (capture the flag) with friends from her college class. One of her team was a penetration tester, and she taught the rest of them as they went. They got to a login screen, and she put "or 1=1 --" into the password prompt, but it was blocked. Bad input. She tried again, using a web proxy instead, and intercepted the request, changing the password field manually *after* the JavaScript input validation. The application accepted it and logged them into the system! The pentester explained that this was an SQL injection and that if the input validation was just in the JavaScript and not on the server, she could get around it with a web proxy. One of the women immediately got up to leave. When Alice asked her where she was going, she said, "To my office! I need to check all of our apps for this! Oh my gosh!" and off she went. They all learned something valuable that day!

When hackers, ethical and unethical, look at our systems, they are looking for weaknesses and ways to get around the security controls we build into our systems. One of the ways they do this is by finding ways to change or replace the system's code with their own code (which is called *injection*), variable values, URLs, session IDs, etc. An opportunity for them to accomplish this is when important functions happen on *untrusted systems*.

What Is an "Untrusted" System?

A system that can be changed by the end user or a third party is untrusted. It is untrusted because we do not control it completely. If a system can only be changed by the owners (you, the programmer) and cannot be modified or accessed by external users or systems, that is a trusted system.⁷¹

To ensure that a system is trustworthy, we need to follow best practices (such as regular patching and following the hardening guide), fix any vulnerabilities we find, do regular scanning to ensure that there are no new vulnerabilities, and usually use a combination of monitoring, logging, and/or a detection system.

An example of an *untrusted* system would be the front end of a web application, written totally in JavaScript, which runs in the browser. Because a malicious actor can use a web proxy to intercept JavaScript requests and responses, this means the values in the requests and responses can be manipulated by the end user. If instead of validating a value only in the JavaScript, we sent it to the server to validate it there, this would be more secure because the server (assuming it is properly patched and hardened) is a *trusted* system. We would control the server; it cannot be manipulated by the end user.

What Are “Important Functions”?

Any security control (a part of your code that performs a security function) is an *important* function; this includes authorization, authentication, password changes, changes to privileges or roles within your system, encryption, etc. Anything that performs a transaction, such as a payment, updating a record in the database, creating or deleting a user, etc., is also an important function. Anything that makes a decision in your system is also considered an important function.

Putting It Together

To ensure that all important functions happen on trusted systems, it is best to make a list of all trusted and untrusted systems that your application will deal with⁷¹ and mark them clearly on your design documents and diagrams. Ensure that authentication and authorization, as well as throughout input validation, are always used for data and requests received from any untrusted systems. Although we should always test each integration our systems have, pay extra special attention when testing integrations with untrusted systems.

NOTE Getting around a blocklist is often a trivial task for a malicious actor. Some websites assist in getting around such security measures, such as the XSS Filter Evasion Cheat Sheet from OWASP,⁷³ which helps penetration testers stretch the boundaries of any input field. Malicious actors also share information with each other about how to attack internet-connected systems better. The more information groups like OWASP, Def Con chapters, Let’s Talk Software Security, and other communities share on how to protect ourselves, the better we can all do when fighting cybercrime. Whenever possible and reasonable, share tips, data, or other information widely so that we can all be better defenders.

Allowlists versus Blocklists

When talking about input validation (and who doesn’t constantly talk about input validation?), a *blocklist* is a list of characters, domains, or anything else the application will *not* allow. An *allowlist* or *approved list* is a list of what *is* allowed.

You might be thinking right now, “Great! I will make a list of all the bad stuff and block it. This will be easy.” Friends, that is what I thought when I was a software developer as well. Then I became a penetration tester, and in about 5 minutes I figured out how to get around a blocklist. I definitely worried about the past apps I wrote when I figured this out.

Why Are Block Lists Bad?

Coming up with a complete list of every single character that might be bad for your application is *hard*. There are obvious ones such as <, >, and maybe '. But what about), {, \, and ~? Are those bad? Would you test every single character to find out? It's easy to miss one, as you would need to do this for every single input. And input can be quite different for each field: think of an email address versus a person's name. One must have the @ symbol and a period, whereas the other might allow single quotes and hyphens. It would take a long time to come up with all the possible bad characters.

Now think about tricky hackers. What if someone enters “%3C”? If you have < blocked, it won't matter because this just got past your blocklist. URL encoding is one of several ways that a bad actor may try to get past your blocklist. This is why we should *always use allowlists*.

How Do We Create an Allowlist?

Let's say you want to create a username field. Generally, the product owner or business representative (the person who has the authority to decide how the product will work) decides how much space should be given for each field. The business analyst (BA) is usually in charge of documenting this. Ask the BA to work with the product owner to set maximum and minimum sizes for every field, as well as a list of approved characters. You might need to join them at first to explain your reasoning; feel free to let them read this section of the book. If they say “It doesn't matter,” you can likely choose the limits yourself (which means you can be very strict!).

If you are put in charge of selecting the acceptable characters for various fields, please use the following guidance to help you:

- The strictest is a–z/A–Z/0–9. This is all letters (upper- and lowercase) plus numbers. You would normally use this for usernames and other fields that for sure do not need other characters.
- Single quotes are generally needed for name fields. For example: O'Connor.
- For an email address, there is a regular expression (Regex) available that is generally used the world over for this purpose: https://owasp.org/www-community/OWASP_Validation_Regex_Repository and <https://www.abstractapi.com/tools/email-regex-guide>.

Secure Configurations

Security misconfiguration has been on the OWASP Top 10 Risks for Web Applications for over a decade,⁷² and with good reason. Often, when systems are breached, it's not actually from an exploitable flaw within the software; it is that the user has configured the system improperly. Sometimes the default settings are quite open, and the creator of the system assumes you will harden it. Sometimes it's that users undo some of the safety features to make part of the application work better. Sometimes people just assume everything's fine and don't check to make sure that their settings are properly hardened. When it comes to malicious actors, it doesn't really matter why a system was configured poorly; it only matters how much damage they can inflict as a result.

Not to oversimplify, but the easiest way to avoid this issue is to follow the hardening guide of every system you implement in your workplace and at home and then check your settings once a year or so. This is not easy work, and it's not fun or high profile either, but it's very *important* work. It's part of having good security hygiene. And just like brushing your teeth every day probably doesn't feel very glamorous while you do it, having a beautiful smile (and all your own teeth) is something to be proud of. It's the same with having good security hygiene and doing those everyday things that ensure you *continue* to keep your organization safe.

If you buy a configurable off-the-shelf (COTS) software product, such as SharePoint or JIRA, it will come with a hardening guide. Follow it! It will help you lock down your systems. Warning: the more you lock them down, the harder they will be to use, which means you will likely have to make some compromises with other teams. Compromise is okay! Just make sure that (1) you don't allow compromises that keep you up at night with fear, and (2) you document what you changed and why. It can just be an email; it doesn't always have to be extremely formal. The reason is that in case there's a breach someday, you can go back and see "Why did we do that?" It will be handy to know in the thick of a security emergency.

Any system you buy is counted in this category, including your home modem or router, your cell phone or smart watch, and any IoT or "smart" devices. If there's software behind it, and if it's made for a consumer, that usually means there's a menu you can make choices in. Look them over, and try to choose the most secure choices you can. If you aren't sure what the settings mean, use your favorite search engine to find out.

TIP If you use social media, you should check not only your security settings but also your privacy settings and all your other settings. They change, sometimes quite a bit. And sometimes the changes feel . . . as though they are intentionally circumventing the protective settings you previously selected. Although this may seem a bit extreme, I have previously deleted my entire Facebook account and then created a new one

(more than once) to hide all of my old data. You don't need to be that "hardcore" about it, but you might want to think about just how much information about you is on the internet, and if you are still comfortable sharing that information now. *Says the lady with thousands of social media followers . . .

TIP If you are in the EU, you may request a full export of all your data from most products (thanks to the GDPR). Going through your own personal information can be a very interesting exercise!

Hostname Validation

When you connect to a server, you want to make sure that it is the server you meant to talk to. We want to connect using TLS (Transport Layer Security, version 1.2 or higher as of this writing) to ensure that the data we send is encrypted. But we also want to ensure that who we are speaking to *is who we think we are speaking to*, and we do this with certificates. When we check a certificate, we want to ensure that the certificate was issued by a trusted CA (certificate authority) but also that it's coming from the host we are expecting.⁷⁴ This is called *hostname validation*.

We strive to verify who we are speaking to, to avoid an attack called *man in the middle*. A man-in-the-middle attack, also known as a *manipulator-in-the-middle* attack, is where someone is listening to your calls. Someone has wedged themselves in between you and your destination and is reading everything that goes back and forth from your system to its destination. This can be part of "sniffing network traffic," but even worse, as every response and request goes through this malicious third party. For obvious reasons, this is really bad; you are exposing sensitive data.

Occasionally, software developers will validate that there is a certificate but forget to check who issued it or that it's coming from the host we expect. Validating that the certificate was issued by a certificate authority and that it is coming from the hostname you're expecting will avoid this attack. When in doubt, throw an error!

Reusable Code

It is good software development practice to write code that can be reused; why re-create the wheel over and over if we already have a great set of wheels? In practice, though, software developers are often rushed to "get to prod" as fast as possible rather than focusing on creating high-quality, reusable functions that might benefit other teams.

Ideally, reusable code should be⁷⁵

- Flexible, such that it allows the code to be easily used within another application.

- Compatible with different hardware, APIs, serverless, and whatever other technologies you may be using.
- Secure. It should be free of any bugs or defects that may affect the security or dependability of the other application.

SECURE CODE LIBRARY

Alice remembers one of her lead programmers explaining that they had something called a “secure code library,” and she was quite pleased with herself. She told Alice they had a template they used for every new project, full of encoders, validators, login screens, password handlers, and more! She said that by pentesting the template and reusable functions, instead of just one app at a time, they could fix the bugs in all their apps across their entire organization. She proudly boasted about how she had saved the company tons of money, all with reusable code! Alice made a note to say nice things to this lady’s boss . . .

Some advantages of reusing code are saving time, saving money, less risk (because you’ve already tested this code, you know it’s safe to use), easier maintenance, and more.

As you might have guessed, there are some challenges with code reuse as well. The biggest one I’ve seen over and over is awareness; just having other teams be aware that they can use the cool new function you wrote is more work than I realized when I started out. It requires a lot of organization, awareness, and organizational support for code reuse to be common, efficient, and not just a one-off.

To reuse code well,⁷⁶ we need to ensure the following:

- Discovery and usage of our code is easy.
- Interoperability: that our reusable code works with all the platforms, languages, and frameworks that are part of our technology stack.
- Strong governance (code review and testing, at the very least).
- Some form of change management to keep track of what was changed, when, by whom, and why. This can be done with version control and good commenting as a minimalist approach.
- Versioning, for publishing changes in the library and to manage how these changes are communicated to users. You will want to avoid automatic updates that may cause outages and try to make the updates as easy as possible to apply.
- Scalability. Your code might be used in both a tiny app and a giant app; it needs to work for as many systems as possible.

If you work somewhere that wants to create a code library or reusable code, help make sure it’s *safe* code by following everything you learned in this book.

Safe System Calls

When we (programmers) want to talk to the operating system, we call this a *system call*.⁷⁷ Our custom applications generally run on top of the operating system, such as Microsoft Windows, Linux, macOS, Android, Apple iOS, etc. If we want the operating system to perform work for us, we need to ask it via a system call.

Operating systems are extremely powerful; they can do pretty much anything software can do. They can do more than our apps can do; they can launch other apps, stop processes, contact other servers, etc. It is extremely important, critical in fact, that we protect system calls from malicious actors.

The most obvious way to protect system calls from malicious actors is to never make any system calls, which is not always possible. Assuming that you must make system calls for your program to function properly, the next option is to never allow any user input to be used to trigger or influence a system call (*never* use input as part of a system call). If you must allow the user's actions or input to be a part of the call, instead of taking user input and copying it into the call, compare the input against an allowlist of input. If it matches something on the allowlist, the call can go forward (with the text from the allowlist, not the user data). If not, then reject it and start again.

Whenever you are making a system call, it should be done on a trusted system (the server rather than the front end) to ensure that you have complete control over the call you are making.

Mitigating Circumstances

On top of this, you should install protective software on your server on which you are making the system call. For instance, your file servers should not have execution abilities. Your web server should not allow new software to be installed on it. All server systems should have anti-malware, intrusion prevention/detection systems (IPS/IDS), application control tooling, and file integrity monitoring (FIM) installed.

Although there is more you can do in this area if you are writing very low-level code, most of the world is writing web apps most of the time. If you are writing low-level software, read the following paper for more ways to protect your system calls: "Protecting Against Unexpected System Calls"⁹² by C.M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S.K. Debray, and J.H. Hartman.

Commenting and Other Documentation

Comments and documentation are your way to communicate to anyone else working on your project, now and in the future. If you are concise and informative, you can make the next person's work a pleasure. The opposite is also true. Let's dive into how to do a great job of this together.

Comments

Comments in our code are an explanation of what we were thinking and what the code is supposed to do. Ideally, we leave comments littered throughout our code to help the next programmer (sometimes ourselves). Often a function will start with comments telling the reader what the entire function is supposed to do and then have more comments as it gets to more complicated parts.

Here are the pros of leaving comments in your code:⁷⁸

- Comments explain to the next programmer how part of the code works and why you wrote it that way. This can lead to code that is easier to maintain and faster bug fixes.
- Comments in the code clarify what you were thinking when you wrote it, ideally making it more pleasurable for the next programmer. It might sound odd, but code that is really difficult to work on does not lead to great employee retention. It can also result in people not wanting to work on projects with you.
- Comments are a reminder of what you were thinking. When enough time goes by, most of us forget.
- Some tools can be used to create documentation from your comments, saving you time (example: pydocs).

Here are the cons of leaving comments in your code:

- If there are *lots* of comments, they may become overwhelming or annoying—for instance, if there are more comments than code.
- Out-of-date or incorrect comments can be misleading. Imagine that you fix a bug but then don't update the comments, and now the comments are incorrect. You could send the next programmer on a wild goose chase. No one wants that.
- Confusing comments can be . . . confusing. Not everyone is a great communicator.

On top of comments, whenever possible, try leaving self-documenting or self-explanatory code. For instance, you can name a variable `math1`, and the next coder will likely assume it's going to be used for something to do with math. That's a tiny bit helpful.

If you wanted to be more helpful, though, you could name it `intAverageHeight` instead to say that the variable is an integer and that it is used to calculate the average height (of whom, we're not sure).

To do even better, you could name the variable `intAverageStudentHeight` to say whose height is being stored (perhaps there will be another variable named `intAverageTeacherHeight`). And your function might be named `AverageHeight`

because it averages the height of any group of people, students, teachers, and anyone else.

Adding naming conventions that clarify what you're up to is a great way to make easy-to-read code!

JENNIFER

Once, Bob was hired as shadow IT. He didn't know it at first; it wasn't his fault! The business unit hired him, and he thought he was the entire programming team. Before him, two other programmers worked there, and one was named Jennifer. The first time Bob opened one of her apps, he understood why she was an ex-employee. All of her variables and functions were named . . . Jennifer! Without even a prefix to say which type the variable was (for instance, `strJennifer` would at least have made it clear it was a string). With functions `Jennifer1`, `Jennifer2`, and `Jennifer3`, and then variables `Jennifer4`, `Jennifer5`, and `Jennifer6`, Bob was completely lost in about two seconds! Bob had to rewrite all of her stuff so that it made sense and could be maintained by other programmers. He remembers explaining to his boss why it was taking so long: he had no idea his boss's eyebrows could go up that high on his forehead.

NOTE Believe it or not, programmers used to avoid leaving comments because then they thought it would be easier to replace them with a different programmer. Programmers (in the 1980s and early 1990s) were occasionally known to make their code intentionally difficult to maintain, to create job security. It didn't work very well.

Documentation

Documentation is a bit trickier. It's very easy to overdo documentation (compared to commenting) and end up spending more time on the documentation than the benefit you receive from it. Documentation's entire purpose is to make the creation, maintenance, and security of your applications faster, easier, and cheaper. If you spend twice as much time keeping your documentation in order, are you still gaining anything? Thus, it's important to have balance in the amount of documentation we create.

Let's start with some of the benefits of documentation for your applications (this can be an architecture document, a testing plan, a design plan, etc.):

- Documentation that is well written gives a fast overview of your application.
- Code documentation provides a high-level overview of the code and how it fits into the overall system. This can be particularly useful for new developers who are trying to understand the codebase.

- Code documentation can include examples and usage instructions, which can make it easier for developers to use the code correctly.
- Code documentation can help to ensure that the code is maintainable in the long term. By providing a clear explanation of the code's purpose and how it should be used, code documentation can help to prevent misunderstandings and reduce the risk of errors.

Here's some documentation advice:

- Be brief and concise. Make your documentation as short as possible while still getting everything important across. You do not get paid per page (generally), and thus writing more does not provide more benefit. Get straight to the point as fast as possible.
- Use clear language to make it as easy to consume as possible. It might be more work while you are writing your document to make it as clear as possible, and you may have to create multiple drafts. But if you can make it twice as fast to read, and hundreds of people will read it, your efforts will save time and money (and perhaps a few headaches, too!).
- Include images, if possible, to illustrate your points (such as your design or flow). People learn in all sorts of different ways, and images help more people understand faster.
- Less is more; only create documentation you believe will be helpful (unless there are policies or regulations that force you to).
- Do not ask for documentation you do not need. Do not ask for specifics in your documentation (such as a list of IP addresses, third-party vendor forms, when their last pen test was, etc.) unless you *need* it. Don't waste people's time with something that you *may* want later. You may be thinking, "I would never do that," but in my 27 years working in tech I've seen quite a bit of documentation requested that was never read, which wasted untold engineering hours.

CLEVER COMMENTS

Alice remembers having to take a programming course in college in which the professor explained the importance of comments. She thought it sounded pretty handy and tried to be good about it. Later in the course, she was asked by another student to look over her code, and Alice was *so lost*. The logic seemed all over the place, and she ended up asking several questions. By the end, she had convinced the other student to add all sorts of comments, and the student got an A on their assignment. Alice felt pretty pleased with herself.

BOB GOES BACK IN TIME

Bob used to think that commenting was just plain dumb: “If the next programmer can’t figure it out, maybe they aren’t a good coder.” He thought this until his second year of coding when he had to fix a bug in some of his own code. It took him 4 hours to figure out what the heck he had been trying to do before he got down to fixing the bug. From then on, he saw the value in leaving comments!

Verification of User Consent

Our users generally* have rights. If the user is a consumer, they can choose to use our software or not. In most countries, you need to ask the user’s consent for various actions you may take in your applications, such as saving, sharing, or copying their private data, deleting data or users, etc. We ask every time to ensure that we do not upset our users. We ask even if there is no way for the user to notice or see that we are doing it, not because we do not want to break the law, but because *it’s the right thing to do*.

You must ask for user consent in the following situations (as a minimum):

- Any form of collection or processing of personal data⁷⁹
- Creating or deleting an account
- Changing a password
- Changing contact info (including username/email address)
- Giving access to others
- Sharing user data with a third-party tool (this includes calling third-party APIs, like Google Maps, and sending them your user’s data in your request)
- If you decide to use data in a different way than the way you originally said you would use it
- Selling the user’s data
- Training your AI on their data

When in doubt, obtain user consent.

NOTE *Sometimes, users do *not* have rights. The Canadian Revenue Agency (CRA), the tax office for Canada, forces citizens to use its services to pay their taxes and be otherwise compliant with tax laws. CRA’s non-internet services are available only during usual working hours and generally involve waiting most of your day on hold, which most Canadians cannot afford to do several times per year (including the author of this book). Thus, most Canadians are forced (to avoid financial hardship) to use the CRA’s online services. Unfortunately, the CRA’s terms of service (ToS) state that it is not legally liable if it loses your sensitive data or you are otherwise harmed by using its digital

systems and services. This is the type of ToS that most of us would normally refuse; why would we give our most sensitive data (our financial information) to an organization that claims it is not responsible if it loses the data? The average person would just use a competing application or service. Regretfully, there is no alternative tax office for Canadians to use instead with ToS that are reasonable and fair. As of this writing, the CRA has lost the personal information of thousands of Canadians in one or more data breaches, including the author's parents, and is under a class-action lawsuit.⁸⁰

Integrity Checks, Code Signing, and Immutable Builds

When the “new” OWASP Top Ten Risks to Web Applications came out in 2021, the nerdy author of this book was extremely excited to see that Software and Data Integrity Failures was #8!⁸¹ Finally, someone pointed out that we cannot always trust that our builds are truly *ours*. In *Alice and Bob Learn Application Security*, we covered that you shouldn't trust anyone, not even your mom,⁸² and I meant it. When we compile our code or build an image, unless we use it immediately, how do we know it's unmodified? By doing an integrity check, that's how!

When we build our code using a CI/CD, the artifact (the compiled app, container, virtual machine (VM), report, etc.) is copied somewhere. Ideally, if it's a container or VM, it's deployed. If it's an app, it's deployed onto one of those containers, VMs, or a Platform as a Service (PaaS). But sometimes it's saved somewhere else to be deployed later, especially when doing waterfall or another non-DevOps programming methodology.

Doing waterfall software development for the majority of my programming career, many times I saw a developer copying and pasting their app to and from different servers. It would have been simplistic to swap it out with another copy, and no one would have been the wiser. This is where an integrity check comes into play.

An integrity check can take many forms. For people, it often involves looking at a person's face and asking difficult questions: “Did you eat that cookie, young lady?” or “Have you had anything to drink tonight, sir?” You can often tell by the look on someone's face and/or their answer whether or not they are telling the truth. Unfortunately, computers aren't as intuitive as people, so they need digital ways to validate the integrity of a build, software, message, etc.

A software integrity check means validating the exact copy of the software you think it is. This can be done via⁸³ validating a checksum or a digital code signature. This type of check is often performed during a software install, after a download (before using what you downloaded), while running your CI/CD, during a deployment, etc.

We can also protect the integrity of our software supply chain:⁸³

- Ensure that the dependencies we use inside our software are not known to be vulnerable (or, if they are, that we are not calling the vulnerable function inside of it and/or it is not exploitable the way we are using it).

- Ensure that the dependencies we use are from a trusted source (a repository known to be good, perhaps one that your organization controls or pays for access to, as opposed to less-controlled repositories such as Docker Hub).
- Ensure that the access to your code creation tools is strictly controlled, such as your code repository, your CI/CD, and all other mechanisms for releasing and deploying software.
- A review process for code and configuration changes can also help in this area, but beware of creating a slow manual process; plan for speed and ease of use.
- Add automated code review, such as a SAST, to your process. (This isn't guaranteed, as advanced malicious actors might still be able to get bad code through, as we saw in the Solar Winds security incident, but it's extremely helpful.)
- If using CI/CD, more than one person must be involved to push code to prod. No single person should be able to get to prod on their own. This is known as *separation of duties* (the same person also should not be able to control the entire process).
- All users for CI/CD and other publishing software must have MFA enabled.
- If you have the budget and your risk profile is high, get your supply chain pen tested.

NOTE Alex Dow of Mirai Security in Vancouver did a terrifying security talk at the SecTor security conference in Toronto, Ontario, in 2021, where he demonstrated compromising a CI/CD DevOps environment over 20 times during a single penetration testing engagement. Luckily for us, Alex is one of the good guys who helps us secure our digital worlds, but his presentation made it clear that it's a lot easier to break into these systems than most of us would like to think. Please take good care of these powerful systems.

Immutable Builds

Immutable means unchanging over time or unable to be changed.⁸⁴ Immutable infrastructure means we deploy it, and that's it. We do not patch it; we replace it with a newly patched version instead.

The main benefit of this⁸⁵ is that we can roll back in an instant. If we patch things live, and it goes badly, there's often no way to roll back. We have to fix it, live, in production. With immutable builds, we have a perfect copy of all previous versions that we can go back to at any moment. And if we patch it in dev and it goes badly, we just roll back. Once we have a working copy that is tested thoroughly, we can copy it into prod.

Another benefit of immutable builds is that we always know *exactly* what is deployed to our servers and what it contains. It's a perfect record of what was there. It also makes for easier replication; no need to manually dig through config files that have been updated or changed when you're trying to create 20 more instances of that container.

Using immutable signed builds is another way that we can help ensure the integrity of our software systems.

Avoiding Brute Force

Brute Force: relying on or achieved through the application of force, effort, or power in usually large amounts instead of more efficient, carefully planned, or precisely directed methods.

—Merriam-Webster⁸⁶

Brute-force attacks against software systems use automation, sometimes called *bots* or scripts. Most brute-force attacks are not very intelligent; or they used not to be. Malicious actors have learned, and some have adjusted against our defenses to make more sophisticated attacks. APIs are often subject to brute-force attacks more often than web applications or “front ends.” Usually, the attacker is attempting to gain access to a system for which they do not have proper credentials, but sometimes they just want to bring down your servers to cause harm. Either way, it's bad news.

The most common way⁸⁷ to prevent a brute-force attack is to lock an account after a certain number of failed attacks. Unfortunately, if you do not do this carefully, an attacker could ensure that thousands of accounts are locked out and create a denial-of-service (DOS) attack in the process.

A second measure that is also common is to use rate limiting and/or resource quotas, slowing down the requests and/or cutting off a user after a certain number of tries. Although this is similar to the first option, it's generally implemented using a technical control outside your application, such as an API gateway, Web App Firewall (WAF), or another network appliance.

Another measure suggested by OWASP⁸⁷ is to fingerprint the device and lockout based on if it's the same browser/IP/other identifying information. You need to track (using a cookie) a browser that was previously successfully used for logging in. If it was used before, perhaps a lockout is not necessary, just a time-out period. This depends on your risk profile.

Other measures include blocking IP addresses from certain geographical locations (perhaps nations that are not so friendly with your nation), IP addresses that are from well-known proxy sites, throttling requests (slowing down requests enough to annoy a computer but not a user), resource quotas (only allowing so many attempts, no matter how slow the attempts are), asking for a second factor of authentication (MFA), asking for another of the same factor (usually

the answer to a “security question,” which is not a security control that I recommend using), adding a CAPTCHA, allowing logins only from specific pre-approved IP addresses or browsers or devices, or providing unique login pages (for very important users, such as administrators). Ideally you would set up a WAF or other tool to perform these checks and blocks for you, dynamically and automatically.

All of that said, enforcing the use of MFA for all users solves a lot of these problems. This doesn’t make sense from a business perspective for every situation, but whenever possible, push for it. MFA is the #1 user-actionable defense against credential stuffing (with the second being using long, complex, unique passwords and/or passkeys).

Security Controls

Security Control: A safeguard or countermeasure prescribed for an information system or an organization designed to protect the confidentiality, integrity, and availability of its information and to meet a set of defined security requirements.

—*National Institute of Standards and Technology (NIST)*⁸⁸

In other words, a security control performs the action of enforcing a security policy. In software, any code that makes security happen is a security control. Examples include forcing users to go through a layer of authentication to log in, asking users for a password, encrypting data in transit or at rest; you get the picture.

Security control categories are⁸⁹ physical, technical, and administrative. Security controls themselves can be preventative, detective, or corrective.

Let’s take a closer look at each, shall we?

- *Physical security controls* are things like locks on doors, windows, or your locker back when you were in high school. They could also be guards, mantraps, or gates. This can include hardware-based controls, such as a hardware security module (HSM) built into a laptop.
- *Technical security controls* are software features used to enforce a security policy, such as an MFA challenge (a password manager comes to mind), a firewall, or anti-malware software/endpoint protection.
- *Administrative security controls* are policies, procedures, guidelines, standards, etc. They are the rules we must follow to participate in whatever system they apply to (for instance, your workplace).

Generally, physical and technical controls are used to enforce administrative controls. Ideally, you also have some form of tracking and alerting (logging, monitoring, and alerting) on all types of controls to ensure that they are being followed.

When researching this topic, you will also stumble on preventative, detective, and corrective controls. You can categorize each implementation of the security controls into one of these types of measures.

A *preventative control* attempts to stop violations before they happen. Locking a door, encrypting data, or creating a policy before something bad has happened would be examples of each.

A *detective control* attempts to figure out if a violation has occurred or is occurring. Any sort of logging, monitoring, and alerting, including a loud car alarm, would be examples of this type of control. A matching administrative control would be a call tree (a list of who to call when an emergency happens) or your incident response triage process.

A *corrective control* attempts to fix or repair the violation or attack. The eradication and recovery parts of your incident response process should cover this, as well as backups and rollbacks for technical, and repairing a broken window or rekeying a lock covers the physical realm.

Other types of security controls include deterrent, directive (policy/guideline), compensating, or recovery.

If you are going to create a formal application security program (or any type of security program), you will need all of these forms of controls in order to have a complete and robust program.

Handling Elevated Privileges

As a software developer, you are essentially a technological superhero. Not only do you have a set of amazing skills that make you extremely powerful, but on the networks and systems you work on, you are usually also granted additional privileges, access, and abilities. As such, your accounts are a high-value target for any malicious actor. If an enemy of your organization could take over your work accounts, they would be able to do an amazing amount of damage to your company. This means you need to protect those accounts and your privileges very carefully.

The two best ways to do this are to use a long, unique, complex password (or passkey) *and* enable multifactor authentication on all of your important accounts. By doing this, you make it extraordinarily difficult for someone to attempt to get into your accounts.

Another thing you can do on top of this is apply the concept of least privilege to all your accounts. If you do not require access to something, perhaps you can give up that access. If you do not require certain tools anymore, perhaps you can give up access to the tools you no longer need. Although giving up access and tools can be limiting at times, ideally, you would only give up things you truly do not need to perform the duties of your job. Think of it like spring cleaning your digital footprint.

On top of the items listed, you should consider using a password manager to manage all your passwords and potentially your second authentication factor. A

password manager handles storing your passwords safely and securely. If you do use a password manager, you should ensure that the one password you use to get into it is extraordinarily long and complex and something no one could ever guess. Mine is a joke. Yes, an *entire* joke. The question *and* the answer. It is quite long because the accounts in my password manager are extraordinarily valuable to me.

TIP Never underestimate your value as a target of malicious actors. If someone asks for your password, may have looked over your shoulder, or has used any sort of pressure on you to gain access to your accounts, you should report it to your security team. Even if it's embarrassing. It's better to come clean immediately than let it go under the radar until it is too big to hide. Reset passwords as a precaution if you are ever in doubt.

Security Maintenance

With any important system, you want to ensure that it's maintained properly. We bring our cars into the shop every so often to have them checked to ensure that all the parts work properly, including the safety features. Quite often, IT security folks can be so overworked that we do not have enough time to get everything done each day that we want to accomplish. As a result, sometimes we forget to maintain our tools properly.

It is extremely important that we check over and test our most critical systems. This means firewalls, web application firewalls, intrusion prevention and detection systems, and any sort of defensive system need to be tested, ideally once a year. When we say *security maintenance*, we mean the maintenance of our security systems or the security features of our systems. We mean ensuring that they're working properly, that all of the access we need is available in case of an emergency, and that they are properly patched and securely configured.

If possible, set up an annual reminder to test the systems you are in charge of. This does not mean you need to do a full penetration test, which would be expensive and time-consuming unless the system is truly critical and/or your risk posture demands this. Generally, you can review the latest version of the hardening guide and verify that you have followed it thoroughly. Check that everyone who needs to have access does and those who no longer need access have been removed; and if there are any "break glass in case of emergency" accounts, be sure those also work. You want to ensure that you are patching all of your systems regularly, but this annual check is a chance to verify that you are up to date and where you think you should be.

Although this is not a sexy task, it is part of having excellent security hygiene. An organization that validates that its systems are up to date, securely configured and patched, and buttoned up tight is the organization with more mature security programs and better security posture. It also tends to have fewer and

less-expensive security incidents. Think of it as an investment in the future of your organization.

Repaying Technical Debt

Many times in my career, I have witnessed what I call “technical debt collection,” generally in the form of a security incident costing a small fortune when basic patching could have prevented it. We take care of the things at home that matter to us, and we should also take care of the things that matter to us (and our organization) at work. Technical debt is the (bad) result when development teams rush a product out the door, knowing full well that they will have to redo some of the work later. The work is sloppy, incomplete, or otherwise problematic, and they know it, *but they ship it anyway*. Situations like this happen generally because of pressure from the business part of the organization.

Technical debt is security debt.

—Dave Lewis⁹⁰

When I first switched to security from software development, I immediately started noticing that technical debt ended up turning into security problems quite often. I would see an application that was successfully attacked via a vulnerability that my scans had picked up 6 months ago. I would see systems that hadn’t even been tested before being exploited mercilessly. I would see systems that were pushed out the door in a giant rush, where several known vulnerabilities weren’t fixed because “there wasn’t time,” only to be taken down in a blaze of glory only a few months later. This is technical debt collection in action, and it’s awful.

Repaying technical debt means going back and fixing the shortcuts we put in place. This can mean fixing vulnerabilities, updating libraries, refactoring code, or even a complete rewrite (but that is an extreme).

We can avoid accruing technical debt in several different ways. One way is to follow good software development practices, avoiding all shortcuts, point finale. That said, refusing to do what your business sponsors ask for can be detrimental to your future employment opportunities and is not always recommended or even possible. Another way we can attempt to avoid technical debt is to raise concerns about it to the security team if we see something happening that we are uncomfortable with. Sometimes this can also work, but not always. Even the security team is powerless at times.

Some organizations will allow you to add notes to the documentation for a project, and you could add notes when technical debt is (knowingly) being accrued. You can not only ask to have your concerns documented as part of the project but also ask for a commitment from the business unit for when this debt will be repaid. That said, this is easier said than done, and often a single

developer will not have enough influence and power in an organization to push something like this through.

If going through your manager and/or product owner doesn't work, sometimes you can repay technical debt stealthily. Several times in the past, I added small feature rewrites, updates, or other changes to larger projects, or I put staff onto such tasks whenever other work had slowed down (such as during summer vacation periods). My bosses rarely noticed, and if they did, I pointed out all that my projects were on time and on budget. As long as this is not work that your team is supposed to be charging customers for, you are unlikely to get into trouble doing it, assuming the rest of your work is arriving on time.

TIP There are code quality tools (sometimes included with linters) that will help you measure your technical debt and keep track of it. Having a tool like this can go a long way when trying to get management buy-in for a serious refactor.

If you work hard to avoid accruing large amounts of technical debt in your organization, your security posture is likely to be better, you will have fewer security incidents, and you might have a less stressful work life. It is generally well worth the effort!

Chapter Exercises

You can find the answers to these questions in Appendix B, "Answer Keys."

1. Give one example of technical debt (real or fiction) and how you could "repay" that technical debt.
2. Name one reason why protecting your work accounts is important. Name one way to protect your work accounts.
3. Give three examples of security controls. They can be physical, technical, administrative, and/or preventative, detective, or corrective. Your choice. They can all be the same kind if you want!
4. Why is avoiding brute-force attacks important? And, in your own words, why is it difficult?
5. In your own words, explain why the integrity of a piece of software matters.
6. In your own words, why does user consent matter?
7. Give one benefit of documentation.
8. Give one benefit of commenting your code.
9. Why do we need to protect system calls?
10. Name one benefit of code reuse. Name one potential problem.

11. Why do we need to ensure that we configure systems securely? What could go wrong? Give one example.
12. Explain what an allowlist is. Explain what a blocklist is.
13. Explain what a trusted system is and why important functions must happen only on trusted systems.
14. Why is readable code important?
15. What is a secure default? What is a paved road?
16. Give one example of a secure default/paved road.
17. Is your CI/CD part of your software supply chain? If so, why? If not, why not?
18. Do we always need to avoid using dependencies with vulnerabilities in them? If so, why? If not, why not?
19. When should we threat-model something, and why?



Summary of Part I

Checklist of General Secure Coding Advice

If you do not have a secure coding guideline or standard where you work, feel free to adopt this document as your first draft. Although this list is not exhaustive, it is written in such a fashion that it should apply to any organization that is responsible for creating most types of software. If you require a very high level of security assurance (banks, counterterrorism efforts, top-secret activities, systems that could cause death), it may be too lax for your organization. Customize and add to it as your organization learns and grows:

1. Assume that all other systems and data your system must interact with are insecure. Validate every input and connection to your systems; reject input and requests that don't measure up.
2. If possible and reasonable, apply the following concepts when designing systems and/or access controls to your systems: *least privilege*, *zero trust*, *defense in depth*, *security by obscurity*, and *attack surface reduction*.
3. Follow and apply any secure defaults or paved roads that have been set out for you by your security team unless there is a business requirement that prevents you from doing so. If so, work with your team to identify an alternative solution, and then document the new solution and reason. Inform the security team of the exception if you have one. They may require further paperwork from you.

4. Design your systems to assume breach. Create plans and processes that plan for failure. Practice for such emergencies, if possible (often called a *tabletop exercise*).
5. It is part of your job to secure the supply chain of the software you build. This means using safe components and validating that the way you are using them is safe, as well as locking down your IDE, sandbox, version control (AKA code repository), CI/CD, your laptop or desktop, and anything else you use to create your software.
6. When designing security features, prioritize usability to ensure better compliance, user satisfaction, and competitive advantage.
7. If there is any error or unexpected behavior during a transaction, always roll back and then fail closed. *Never* fail open. Never attempt to recover a transaction part way through. Always roll back and start again.
8. Ensure that you follow any compliance rules, laws, and regulations that apply to your organization, data, customers, and country(s) of business operations. Ask your security and legal teams for clarification if you are ever unsure of your obligations.
9. If your organization follows any security frameworks, ensure that you follow them, too. If your organization does not, you may want to consider adopting one, such as the guidance outlined in this document and/or book.
10. Whenever possible, share lessons learned widely. It's okay to make a mistake. It's not okay to stumble over the same mistake repeatedly. Be brave and share your failures with your colleagues if you are able.
11. Be wary of backward compatibility. Try to balance both usability and any risks that it will introduce. Document your decision(s).
12. If you have the opportunity to join a threat modeling session, do it. You will not only help make your application(s) more secure but also learn new things, and it might even be fun. Learning threat modeling on your own is a worthy activity.
13. Whenever possible, try to keep your frameworks and dependencies up to date and your systems fully patched. If possible, work toward simplifying and improving the patching processes where you work. This is part of your supply chain.
14. Retest bug fixes and new features for security issues. Regression testing isn't just for usability. Automate as much of this work as possible using custom SAST rules and/or unit tests.
15. Follow a secure system development life cycle (S-SDLC). Ideally, your organization has one set out for you already. If not, follow the advice in Part III of this book to create your own. It is likely that your security team

will not object to you adding additional security activities to your SDLC, assuming they are not potentially damaging (specifically using DAST, a web proxy, a fuzzer against production systems, or arranging a penetration test without their approval). Most other security activities will be allowed, generally without supervision or approval. If you do not have a security team, talk to the head of IT or your boss.

16. If you are allowed to select the programming framework for a project, select a modern framework and use all available security features within it rather than writing your own security controls. Always use the latest stable version of any framework.
17. Perform input validation on all inputs to your applications that come from untrusted sources. If you must accept potentially dangerous characters, escape them rather than sanitizing (removing) them, if possible. Unless you are certain the input is trustworthy, assume it is not and treat it as such.
18. Perform output encoding on all output to the screen and, ideally, also data you are sending to other systems. This helps to avoid cross-site scripting.
19. Always use parameterized queries when making requests to any database, as this forces your parameters to be treated as data, never as code that could be executed or interpreted by the database server. This applies to SQL databases and non-SQL databases alike. Ideally, also use an object relational mapping (ORM) framework, which will make it even easier to avoid database injection attacks.
20. Use a premade system, product, or feature from your framework for managing authentication, identity, session management, authorization, and access control whenever possible. Writing these features is extremely complex and leaves lots of room for error. If you feel tempted to write your own, think of how many security problems there have been with Active Directory over the years and that Microsoft has approximately 1 million employees and contractors, measures its profits in trillions, and is one of the oldest and most-respected software companies in the world. If even *Microsoft* finds it difficult, maybe you should leave this one to the experts.
21. Use a secret management tool to store your system's secrets; *never* store secrets in your code. Use a secret scanning tool to find, remove, and rotate any secrets that might already be in your code base. Then use that same tool on every new code check-in, to ensure that no one ever accidentally checks a secret into your code repository again.
22. User passwords should be salted and then hashed before storing. Peppering is an optional additional layer of security you can add, depending on the level of security assurance your organization needs. Performing hashing

more than once (work factor) can decrease your chances of a rainbow table attack and is generally recommended. Salts are not secret, but they are unique per user; peppers are secret, but all your system's users will use the same pepper value. Store your pepper in your secret management tool, and store your salts in the database.

23. All connections to your web-based systems should be via HTTPS (encrypted) and should use unbroken algorithms, large keys, and follow the cryptography standards of your organization. If your organization doesn't have a policy or standard on this topic, look to the advice from OWASP, the OSSF, NIST, or your government for best practices. If reasonable, select the one that is the strictest.
24. Classify and label all data to make protecting it automatable, easier, and to signal visually how it should be treated. Sensitive data should be encrypted in transit and at rest and, ideally, sent in requests and responses via a secure cookie (or another method that you are certain is safe). Document sensitive data flows, and ensure that these flows are tested for security. Follow any other applicable policies set out by your organization around protecting sensitive data.
25. Use all applicable security headers. For more details on which ones to use and how to configure them, review Chapter 2 or download a cheat sheet here: <https://newsletter.shehackspurple.ca/c/securityheaders>.
26. Use all the applicable security settings for your cookies, especially the `secure` and `httponly` flags. Cheat sheet here: <https://newsletter.shehackspurple.ca/c/cookies>.
27. Access control should be validated on every single request. This is true even for APIs. You should be very diligent when checking that users are authorized to call every page, feature, or record. Have your code that handles this reviewed and tested, especially your API calls.
28. All errors should be caught and handled properly. They should be logged and, if appropriate, alerted on. Cheat sheet available here: <https://newsletter.shehackspurple.ca/c/errors>.
29. Database security entails protecting the database servers (host system), the networks they live on, the database software, *and* the data they house. All four aspects must be hardened, tested, documented, patched/updated, monitored, logged, and protected.
30. It is part of your job to protect the files that are part of the systems you design, create, and maintain. This means your source control, as well as the files your app shares, saves, or uses. Use careful (monitored) access control, exercise extreme caution with files from untrusted sources, ensure that files are properly backed up, and store them in an encrypted format.

31. If possible, always use memory-safe languages. If you must use a language that is not memory-safe, to avoid buffer, stack, string, and integer overflows, do the following: perform careful input validation, avoid using functions that are known to have bounds/overflow issues, perform security testing that focuses on these areas of your code, and follow the rest of the advice in Chapter 3 on this topic.
32. Never deserialize objects from untrusted sources. If you absolutely must do so, follow the countermeasures set forth in Chapter 3, plus advice from the OWASP Cheat Sheet on this topic. Exercise extreme caution.
33. Design and work on systems that respect the privacy of your users, citizens, customers, and employees. If you do not agree with what your organization is deciding for the privacy of your users, ask questions, speak up, object. Use your voice on behalf of your users. In a world where marketers spy on the most intimate moments of our lives, I assure you that your users will thank you for it.
34. All applications should have logging, monitoring, and (when appropriate) alerting enabled. Even your APIs!
35. Avoid potential race conditions by locking resources before using them. Note: most modern programming frameworks perform this task for you!
36. Enable copy and pasting to the password fields in your applications to allow and encourage the use of password managers by your users. This includes all password fields, even if you make the user repeat the password. Disabling copy and paste makes users enter values manually, which then forces them to use less complex, shorter passwords, decreasing the security that the passwords provide.
37. Use strongly typed languages, or ensure that you always define your types and stick to them throughout your entire application. Do not apply poetic license to your variables; ambiguity is the enemy of security and reliability in software. If you choose a weakly or dynamically typed language, test it thoroughly for potential type ambiguity issues.
38. Domain-driven development is a way of designing objects that helps with security, and if you have a chance to use it, enjoy! Although it is not that popular, it is certainly worthy of your consideration as it is very intuitive.
39. To ensure that your designs are secure, perform threat modeling and design review, and apply secure design concepts and design paradigms (processes that help us design). Then hire an expert to perform manual security testing (penetration testing or a security assessment).
40. Whenever possible, provide secure defaults for settings, configurations, and other choices for your users. If you set the default to something very

secure and/or very private, it is likely that the user will leave it that way, meaning they will have better protections.

41. Always try your best to create readable (intuitive and well-documented) and auditable (adding comments to explain security controls) code.
42. Ensure that important functions happen on trusted systems (i.e., the server). Don't do important things in the front-end JavaScript, which could be circumvented with a web/intercept proxy.
43. Always use an allowlist over a block list when performing input validation. It is trivial for most malicious actors to get around a block list, and it is much easier to maintain an allowlist.
44. Verify the security configurations of any systems that you are responsible for, annually or more often. Follow the hardening guide. If they are important enough, perform a penetration test as well. Never keep default passwords; always reset them to something long, complex, and unique. Always run system updates within a reasonable amount of time (do not wait weeks or months), especially ones that include security fixes.
45. When connecting to other systems across the internet or any untrusted network, validate the hostname by checking that the certificate is valid and from the host you are expecting. This is an integrity check.
46. Whenever possible, write code that is reusable. If you are going to reuse code, perform security testing on it before releasing or reusing it. Over time, continue to perform security testing on it, and update all places it has been used whenever a security bug is found and remediated.
47. Always exercise extreme caution when performing system calls, and, if reasonable, avoid using them. Never send user input directly to a system call.
48. Leave helpful comments in your code; it makes the code easier to maintain and understand. Document your applications and design decisions. Both comments and documentation help with most forms of security testing and sometimes incident response as well.
49. Verify user consent and document it (in the database). Do not assume consent; always ask.
50. Performing integrity checks can prevent numerous types of attacks, and it is worth your time to do them. Code signing and immutable builds are two methods you can use to ensure the integrity of your systems.
51. Avoiding brute-force attacks on login functionality is always a careful balance of protection versus usability. Work with the business to come up with reasonable limits and with the security team to monitor and respond if it seems as though an attack is in progress. Carefully log all login

activities (successes and failures), and program your app to send an alert if limits are reached that indicate there may be an attack in progress. Test thoroughly.

52. Your (application) security team is likely to ask you to add security controls to your software projects. If they do, add them. As a reminder, technical security controls come in three flavors: preventative, detective, and corrective.
53. Use all available preventative measures to protect *your* accounts that have elevated privileges, including turning on multifactor authentication (MFA); using a long, unique, complex password; and using a password manager. Register with haveibeenpwned.com as well, for good measure. Reset passwords immediately if you suspect a breach.
54. Technical debt often creates security debt and thus, reduces the security posture of your organization. On top of that, it makes it more difficult to respond during a security incident if you are trying to get a bug fix out or update your dependencies. Working hard to avoid or “repay” technical debt will not only make your applications easier to maintain but also help you keep them more secure!

Although this list is long and full of difficult tasks, the work you do to follow it will help you create tough and rugged applications that are difficult to attack. It will also protect your organization and make you a better programmer. I wish you great luck on your journey toward writing more secure code!

Sincerely,
Tanya Janca

Part



Specific Advice

In This Part

- Chapter 5:** Technology-Specific
- Chapter 6:** Popular Programming Languages
- Chapter 7:** Popular Frameworks
- Chapter 8:** Vulnerability Categories

Technology-Specific

In this chapter, we're about to embark on a security adventure that covers not just one, not two, but five fascinating topics: APIs, mobile apps, web sockets, IoT devices, and serverless. Get ready to fortify your applications, because we're diving into the exciting world of securing these modern technologies!

By the end of this chapter, you'll have an arsenal of security knowledge to make these types of applications tough, rugged, and, best of all, *secure*.

API Security Best Practices

In this section of the book, we will cover Web API security best practices, focusing on the RESTful APIs, using the OpenAPI protocol rather than SOAP (which is losing popularity)⁹³ or GraphQL (currently gaining popularity but still only a small percentage of API use worldwide).⁹⁴ SOAP and GraphQL are not covered in this book.

NOTE Several of the items in each section apply to all the other sections in this chapter. They will be denoted with a *. Apply these items against all technologies, not the section you are reading.

APIs are apps that perform services for and communicate on behalf of other apps and are often paired with a front end or some sort of API documentation

file (such as an OpenAPI definition file). Although APIs technically have no front end, they still face constant attacks on the web and are a newer area of focus for attackers.⁹⁵ As APIs are often connected to our data and internal systems and are systems in and of themselves that we depend on, we must protect them just as thoroughly as any web application or other piece of software.

Two prevalent threats to APIs are bots (automated scripts on the internet that are constantly attempting to brute force systems or perform denial of service [DOS] attacks) and broken authorization (access provided by mistake to data, functions, systems, etc., that should not have been granted). In the OWASP API Security Top Ten 2023, broken access control is arguable in 4 of the 10 spots; it is a serious problem. In this section of the book, we will discuss best practices for avoiding these threats and numerous others, as well as how to build well-maintained, dependable, and rugged APIs.⁹⁶

Best practices:

- The first thing you should do when securing your APIs is to conduct an inventory exercise⁹⁷ to ensure that you are giving every API endpoint (including actions against the endpoint) security attention, not just most of them. You should include documentation about how they work, the types of requests they accept, limits, etc. Link the documentation to the inventory so it's easy to find.
- Up next, you want APIs to only be accessible via an API gateway. Use the API gateway to enforce authentication and first-level authorization, as well as rate limiting, throttling, and resource quotas. You should also consider turning on input validation if it's a feature in your specific API gateway.⁹⁷ Note: you still need to do your own input validation; the one from the API gateway is not enough on its own.
- Add more layers of authorization/access control to your code beyond what your gateway offers, and thoroughly test it. Validate access on every single page and every single function call.⁹⁹ Create an access control matrix to show which functions and features each user is allowed to call, and then manually test against the chart (automated testing is never enough for this type of functionality). Repeat testing after each major release or any changes to the way you handle this functionality in your system.
- For authentication, use existing systems (OAUTH or JWT version 2 and above); do not write your own (it's so easy to get wrong).⁹⁹
- Use rate limiting.⁹⁶ Set a limit on anything you can. Set resource quotas as well. If there is anything that can possibly have a limit, add one. Examples:
 - Execution timeouts
 - Max allocable memory
 - Number of file descriptors

- Number of processes
- Request payload size (e.g., uploads)
- Number of requests per client/resource (this is also called resource quotas)
- Number of records per page to return in a single request response
- Ensure that you follow secure coding practices (specifically, input validation, input sanitization, escaping, output encoding, and parameterized queries).
- Encrypt data in transit (and at rest!).
- Use defense in depth (multiple layers) for authentication to avoid brute force (rate limiting) and credential stuffing (threat feeds,⁹⁷ stopping users from using known spilt credentials, forcing password resets when credentials are spilt).
- Only use service accounts to connect to other systems; do not use programmer/human accounts. Set up monitoring and logging of the service accounts to look for suspicious behavior.
- Ensure that secrets are handled programmatically and stored in a secret management tool, *not* in your code.
- Don't return extra data that is not needed, with the plan to sort it out on the client side. Attackers are aware that programmers do this for ease of use and exploit it mercilessly. If the data never leaves the server, you don't need to worry about protecting it in the front end. Apply least privilege to your data.⁹⁷
- Set your API web application firewall (WAF) to have a maximum response size.⁹⁸ A WAF is used to filter requests to and from web apps to protect them against attacks. There are WAFs created just for APIs that filter requests to and from APIs to remove potentially harmful input or inappropriate output (for instance, you probably don't want to return the entire contents of your database). Set it a bit above whatever the maximum API response was from the previous month (after validating the data looks appropriate). One megabit for an image is small, but 1 megabit of JSON is a lot of data and might not be what you intended to return to your user. This can help you stop the exfiltration of sensitive data, also known as a data breach.
- When performing authentication,⁹⁹ follow best practices for logins (use hashed and salted stored passwords, offer multifactor authentication [MFA] to users, monitor for breached credentials, force strong and/or long passwords), and block brute-force attacks by monitoring failed login attempts (both fast and slow attacks).

- Protect your data further by classifying and labeling it so it is always treated with proper precautions.
- Protect your API parameters⁹⁹ from user changes (attacks) by using strong data types (integers or Booleans, as opposed to “fun (x, y)” or some other made-up type) and rejecting any poorly formatted requests (even if you are capable of processing it, don’t). Do not accept any parameters that your function does not ask for. (If you receive extra parameters, reject the request altogether; this is an attack. Do not try to process it anyway.)
- Avoid verbose error messages that contain sensitive information. System information, crash data, and stack traces should never be included in any error messages. Be vague; only a computer should read these messages. If your system throws a system error, you should have clearly predefined error responses ready rather than passing through an error object and creating your custom error message from its contents.⁹⁹
- Use an API-focused linter (ideally with an IDE plugin)¹⁰⁰ and/or static analysis IDE plugin that can help you find security bugs or missing security controls in your code.¹⁰¹ Using one that adds “little red squiggly underlines” to tell you where you’ve made an error and then offers code suggestions makes it easy to write higher-quality code. Use both for great results.
- Be careful to ensure that you decommission old APIs properly and completely when you release new versions. Old versions of APIs should never be left running or connected to other live systems if they are not being monitored and maintained. You can leave old systems running for backward compatibility, but you must manage them safely and actively.
- Encrypt all API requests and responses (not just the data). Send everything over HTTPS, never HTTP.⁹⁹
- Continuously improve and verify the security of your APIs.⁹⁹ Run weekly software composition analysis (SCA) and static application security testing (SAST) scans of your APIs in your code repository, and dynamic and/or interactive tests of your live APIs. If you find a serious bug in one API, validate that it is not present in the rest of your APIs, check your logs to see if it was previously exploited, and then create a guardrail using SAST custom rules to block it from happening again.
- Ensure that the design of your system is secure by performing threat modeling on it and applying secure design concepts, as previously mentioned in section 1 of this book. Finding business logic flaws is often a manual process; a good pen tester can also help you do this.
- If your systems are consuming third-party APIs, ensure that you perform input validation, input sanitization (if you must accept potentially dangerous characters), escaping (if applicable), and output encoding on all

data you receive. Ensure that it is what you are expecting, including the format and type, and if it is not what you thought you were getting, reject it unceremoniously.⁹⁶

- You should also¹⁰³ use separate API keys for each app, delete old API keys, manage keys in a secret management tool, always rotate a key that is spilt, and audit usage over time to ensure that none of your keys are being misused.
- To avoid server-side request forgery (SSRF), a new item on the OWASP API Security Top Ten,⁹⁶ only accept URLs from a list of approved URLs (an allowlist), thoroughly test code that performs parsing of URLs, only allow access to any internal services *via* services accounts (blocking your app from asking for resources from the incorrect/unexpected servers), and apply the concept of zero trust when designing your network (blocking all ports/service accounts/etc., by default unless there is a business requirement to allow such a connection).¹⁰²
- Logging, monitoring, and alerting should be turned on.⁹⁶ Ensure that you protect the logs and treat them like the sensitive and valuable data that they are. You also need to double-check to ensure that you are not logging sensitive data, such as personally identifiable information (PII), personal health information (PHI), or complete credit card numbers. If you want to be proactive, perform threat hunting as well by looking through your logs for unusual and suspicious activity.
- Block all unused HTTP methods (sometimes called *verbs*) such as HEAD and TRACE. Most APIs use GET and POST, and occasionally DELETE, PUT, or UPDATE. The other methods that are not being used should be disabled.
- If you have webdav enabled in production (it offers even more HTTP methods for use during development), *turn it off*, please. It is not meant for production, and it gives attackers several more HTTP methods to use to try to attack your APIs. This is an application of least privilege.
- Use a service mesh if you have a *lot* of calls to your APIs and you are experiencing latency or problems with your calls reaching their destinations. A service mesh is not required if you have only a few APIs, but if you have hundreds of thousands of calls per day, you probably want one. As a bonus, it can perform end-to-end encryption without any code changes!
- Create coding standards and other security guidance documents for the APIs at your organization, ideally with items from this section of the book. Make sure they dictate which API protocols are allowed and not allowed (for instance, OpenAPI is allowed, SOAP and GraphQL are not allowed, if that is your policy). Distribute them widely, socialize them, and teach them to your teams.
- Follow all the items from section 1 of this book that apply to APIs!

TIP I don't usually name products in my books, but these are free, so I hope you don't mind, dear reader. 42Crunch's IDE plugin is a semi-free linter for APIs that I absolutely love; it's so easy to use! It will walk you through hardening your OpenAPI definition file to make it rugged and tough. Adding Semgrep's SAST IDE plugin and connecting it to the free Semgrep Community tool will help you check the rest of your code, using the little red squiggles to tell you if you've made a coding mistake in the rest of your API code. If you have little to no budget, these plugins are a great (free) place to start your API tooling journey. I will list even more free tools to help you throughout the rest of this book.

POLITE ERROR MESSAGES

Bob got to sit in on a penetration test once, and it was exciting. Well, it was at first. The pen tester was sending requests to and from the new API that Bob's team had made, and he felt proud while he read the very polite error messages the system returned when the pen tester sent it bad requests. After reading a couple of messages, the pen tester said, "Thanks to these helpful error messages, I was able to fix my attack syntax and make a successful injection attack!" with a big smile.

Bob was taken aback; how could that be? The pen tester showed him the progression of requests that she sent. The first one was very poorly formatted, but with help from the error message, the second one was really close. After four or five tries, she injected her own code and made the database go to sleep for 30 seconds to prove it had worked. Bob is Canadian and prides himself on having good manners, and he thought that having detailed and helpful error messages was part of creating a high-quality application! The penetration tester explained that APIs are called *by machines*, and humans should only see these messages during development and testing. The rest of the time, only computers should see the messages, unless there is an attacker involved. "You don't want to be polite to an attacker, right, Bob? You don't want to help them. So, make your error messages give nothing away. Bad input; try again. That's it." Bob was shocked. He felt so silly! When he later explained to his team, he saw the light-bulbs go on for each of them, and they never made that mistake again.

NOT-SO-DYNAMIC DAST

Alice oversaw the security for one of the systems she commissioned at work. They asked her if she wanted a penetration test or an automated tool to test the APIs her team was working on. The vendor's friendly salesperson told her that a dynamic application security testing tool (DAST) could test her APIs just as well as a pen tester, and she could use it to test all the APIs in her entire organization. Alice felt smart, buying a system that would test them all, instead of paying a lot to have a pen tester test only one of them.

When Alice's team tested the first API, it came back perfect, with no security bugs. Alice was elated that her team had made such a good API! They tested the second one, and it passed, too. Alice was super proud of her team and went home for the day in a great mood.

A few days later, the person in charge of the DAST scans came to Alice with an apprehensive look on his face. “Alice, I’ve tested 27 of the APIs, and *all* of them came back with *no* bugs. I’m not sure how to say this, but . . . that can’t possibly be right. We just started building APIs, and we had no security training, and I don’t mean to sound negative, but I think the tool might be broken.” To be continued...

COSTLY FALSE PROMISES

Alice called the DAST vendor and asked why the DAST hadn’t found anything in her API. They told her that her API definition file wasn’t properly formatted; she just needed to make some small code changes. Alice and her team went back and forth with the vendor, changing their API definition/schema file over and over to try to make the DAST able to talk properly to the API. After quite a lot of work, it finally found a couple of things, but the developers were upset at having to change so much code “just so her janky tool could run.” When Alice added up how much engineering time they had spent, she realized she had taken up almost all the time she had allocated in her project schedule for security bug fixes! And there was no way she could afford to do this much work on all the other APIs just so she could run the tool. She ended up asking for extra budget, which made her look bad, and she hired a pen tester after all. They found several things wrong that the DAST had missed and wrote up a very helpful report. The vendor kept insisting the entire time that the tool “works with APIs” despite the terrible results she got. Alice felt embarrassed to have believed the vendor’s promises without verifying first that the tool did what she needed it to do. Her project ran over budget and was 2 months late. How humiliating! From now on, she would do a proof of concept (POC) process, insist that they install the tool, test it thoroughly, and ensure that it worked as advertised before she signed anything.

BEERS AND BUGS

Bob is an extrovert *and* loves beer, which means he will go for beers with pretty much anyone if invited. One day he went for beers with a pen tester who told him he had been testing version 2 of an API that day and, on a hunch, had called out to version 1 just to see if it would answer. It had! He then started testing it, and it was a security disaster. He found *so many bugs*! He told the manager who hired him, and they were super impressed! He told Bob that was his “secret weapon”: always looking for earlier versions of every API he tested because if there was one, it was almost always a big deal. Bob made a note to go check that all of *his* team’s old versions of APIs were turned off! What a great tip!

This section covered best practices for not only APIs but also for many of the other types of systems we will cover in this chapter. Please remember to apply the items with a * to all the technologies in this chapter, not just APIs!

Mobile Application Security Best Practices

When we're discussing mobile apps here, we will discuss native and hybrid mobile apps, not web apps that pretend to be a mobile app but are basically just a shortcut on your phone to a web app on the internet (known as progressive web apps [PWAs]). As we go through this section on mobile apps, please remember that a large part of mobile functionality is provided by web APIs, and apply what you learned previously to the APIs your mobile apps rely on.

Best practices:

- Mobile apps are often decompiled as part of reverse engineering (to steal your IP or learn how to attack your app), so obfuscate your code and minify it to make this as difficult as possible.¹⁰⁴ There are tools on the market that can do this for you; doing it yourself can be a little tricky to get right. If you do a bad job of it, it's easily reversible, and then it wasn't worth doing in the first place.
- Use an emulator¹⁰⁴ or real devices (a collection of several different types of phones) to see how your app will work on various devices. Test it on several devices—as many as you can. This applies not only to security testing but to every type of testing for better interoperability.
- Storage for a mobile app can be quite different than for a normal web app. We often want to store the data on the phone, for decreased usage fees for our users so we don't make the user re-download the same content over and over, and to reduce latency (the time the user has to wait while the data downloads). With this in mind, validate with the security team (1) that it's okay to store data locally on the phone, and (2) how they would prefer that you do it. Always store data in an encrypted format. Test the functionality thoroughly to ensure that it is safe from user meddling.
- Be careful not to store any sensitive data with or in any of the following:¹⁰⁴
 - The application log
 - Third parties
 - The keyboard cache
 - The IPC mechanism
 - The user's device during interaction
- Set your locally stored data to autodelete¹⁰⁴ after a certain period of time based on business requirements and your organization's information security policy.
- * Never store your encryption key with the data you encrypted it with.¹⁰⁴

- Only ask for device permissions that your application needs to function. Sometimes developers ask for all sorts of things because they can, but this also means that if our application is compromised someday, those permissions can be used against our users. Limiting what we ask for means limiting potential damage in case of an incident (and is an implementation of the concept of “assume breach” that we learned earlier).
- If it is within your budget, deploy tamper-detection technologies that will alert you if someone is attempting to attack your mobile app.¹⁰⁵
- Sessions last significantly longer on mobile than on most desktop apps,¹⁰⁵ and thus, session management is of paramount importance. To protect your session, follow the best practices outlined in section 1 of this book, plus
 - Use tokens instead of device identifiers to identify sessions, which can be revoked in case of lost or stolen devices.
 - Enable remote wiping of data.
 - Enable remote logoff, and kill the session as part of that functionality.
 - Enable automatic session termination if the user does not initiate a logout before they close the app.
- Encrypt any files that you store locally on the device, including your source code.¹⁰⁵ Always use the “latest and greatest” (current industry standard) when it comes to encryption.
- Use code signing certificates to sign your code. This ensures the integrity of your application code to the end user.¹⁰⁵
- * Only connect to APIs after you’ve authenticated and authorized.¹⁰⁵
- * Different areas of your app are likely to have different levels of authorization. Validate authorization on every page, feature, and/or function (depending on sensitivity).
- * If you can afford to implement biometrics or passkeys or use a third-party authentication service, this is generally more secure than implementing your own username + password feature. Use one of those instead.¹⁰⁵
- * Offer MFA or two-factor authentication (2FA) to your users if you can afford to do so.
- *Securely* store sensitive data in RAM by erasing it after use.¹⁰⁵ Validate that it’s actually erased and not still sitting there unallocated with the pointer removed (easily collectible by a malicious actor). Be extra careful with encryption keys!
- Be careful with storing anything in memory on Android devices, as memory is not treated the same way on Android mobile devices as regular web servers.¹⁰⁶ You may want to copy over the flash memory with garbage to

ensure that it is, indeed, actually erased. Note: this may cause slowdowns of your app and phone; be careful and test thoroughly.

- Avoid using a universal unique identifier (UUID) to identify an account, as it's assigned to a phone for life, but phones switch hands.¹⁰⁷ This is a privacy *and* security issue.
- * Be careful with the tracking and processing you do of geographical data. Don't be a stalker: only track the data you *need* to (as previously discussed). Also be aware that this tracking information enters your telemetry systems, logs, and potentially even more systems. Try to balance user privacy with business needs.
- Combine and compile your app's settings data *into your code* rather than into a locally saved file such as a plist, XML, or SQLite file. External files aren't always properly protected by phone operating systems.¹⁰⁸
- If the data in your app is sensitive, ensure that it is not part of backups for the device (unless your app has encrypted it before backup). Otherwise, you are creating your own data spill.¹⁰⁹
- * As per previous advice in this book, do not cache sensitive information from your app on the phone.

IOS-specific advice:

- Disable the auto-correct feature for any sensitive information, not just for password fields. IOS records all non-numeric keystrokes (keyboard cache) to try to help with auto-correct, and this is a security issue for sensitive data.¹¹⁰
- Disable copy and paste for sensitive data other than the password (to allow for password manager use).
- Avoid using WebViews and NSString to avoid framing and clickjacking.¹¹¹
- Use the keychain functionality very carefully, including choosing your protection class carefully and avoiding iTunes backups.¹¹²
- Block caching of application snapshots, which is similar to caching sensitive info.¹¹³
- Implement memory-safety features to avoid overflow-type attacks, including Automatic Reference Counting (ARC), Address Space Layout Randomization (ASLR), and Stack-Smashing Protection.¹¹⁴
- Explicitly disable caching by setting the `cachePolicy` property of the `NSURLRequest` to disable the caching of HTTP(S) requests and responses.¹¹⁵
- Implement App Transport Security (ATS) to ensure secure connections with your backend servers.¹¹⁶

Android-specific advice:

- Set your app's file permissions carefully to ensure that they cannot be read outside of your app, plus encrypt them. For example, do not set something as "world readable" unless you're *absolutely sure* that it's a good idea. If you do, document it.¹¹⁷
- Use "intents" and "broadcasts" very carefully, check your "activities," and follow "webview" best practices. It's easy to make a mistake with any of these Android features; use with caution and test thoroughly.¹¹⁸
- Interact with content providers very carefully. Don't give them write access unless you need to.¹¹⁹
- Protect your application services from external callers by locking them down with all available features (especially setting permissions and verifying them).¹²⁰
- Avoid storing cached camera images if the image may be of a sensitive nature. They get stored in the NAND memory.¹²¹
- Avoid caching GUI objects, such as sensitive images. If you must do so, be careful to fully quit your app afterward (to wipe the cache) or otherwise ensure that it is no longer cached after you end your session or log out.¹²²
- Sign your Android APK with a valid certificate.¹²³

On top of all this advice, I suggest you read the OWASP Mobile Security Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Mobile_Application_Security_Cheat_Sheet.html.

SUPER SCRIPTER

Once when Alice was acting for the head of security while he was on vacation, there was a big security incident. Alice got to sit in on a bunch of meetings and hear all the secrets of what was happening. In one of the meetings, an incident investigator said, "The code was obfuscated," but she wrote a script to de-obfuscate it. Alice asked her how long it took, and the investigator said she did it in between two meetings. Alice was astounded! The incident investigator said it "wasn't advanced obfuscation," and "it was simple." Apparently, their attacker wasn't that advanced, and within a couple of hours, they had sorted out what had happened. Alice was so impressed with the team!

WebSocket Security Best Practices

WebSocket is a computer communications protocol providing simultaneous two-way communication channels over a single Transmission Control Protocol (TCP) connection.¹²⁴ It's generally used to provide real-time data, such as breaking news.

Although HTTP is stateless, web sockets do have a state but less overhead. Web sockets are usually used for things like chat/messaging; real-time, bi-directional transmission; or “constant updates,” whereas HTTP tends to be used to deliver static content or access REST APIs.¹²⁵ Web sockets can also send data in both directions over one connection at *the same time*.

Best practices:

- Apply everything you have already learned in this book.
- Use Secure WebSocket Protocol (WSS) to ensure that your connections are encrypted.¹²⁶
- Implement rate limiting to ensure that you don’t overdo it, like with APIs. This will ensure that you don’t get a ridiculous cloud bill and will stop all sorts of resource attacks.¹²⁶
- Be careful of potential hijacking attempts, and use Same-Origin Policy (SOP) or Cross-Origin Resource Sharing (CORS) security headers to avoid this.¹²⁷ Neither of these is on by default; you need to enable them intentionally.
- Validate the original WebSocket requests, and continue to check that they are from the same place by checking the origin header and passing a token back and forth (just like for Cross Site Request Forgery CSRF) to avoid cross-site web socket hijacking.¹²⁸
- Avoid tunneling through your web socket to other services, such as a database connection. When you do this, if your web socket ends up being vulnerable to cross-site scripting (XSS) or similar issues, all your tunnels are potentially exposed. Instead, access them outside of your web socket (HTTPS, anyone?).¹²⁹
- Along the same lines of rate limiting, limit your payload size to avoid various types of potential attacks. You should know your approximate max payload size and drop connections on anything above that.¹³⁰
- Decide the format (protocol) of your messages to help you validate data from the server and client. If you choose a format, such as JSON, then you know data should always be in that format. If it’s not in that format, then it’s an attack, and it can safely be blocked or ignored.¹³⁰

Serverless Security Best Practices

Serverless computing is a cloud computing execution model in which the cloud provider allocates machine resources on demand and takes care of the servers on behalf of its customers. It provides backend services on an as-used basis,

allowing developers not to have to worry about the infrastructure or servers the app sits on. When a serverless app is triggered/invoked, it starts up a container, launches the app, runs to completion, and then self-destructs, releasing all the resources back into the cloud.

Best practices:

- * Just like any application, only grant your serverless apps the permissions they need to get the job done (least privilege) and be very careful to ensure that your identity and access management are implemented carefully and tested thoroughly (identity and access management [IAM]).¹³¹
- Pick a modern and up-to-date environment or language for the serverless function you create.
- Put it behind an API gateway, and configure the gateway to perform authentication, authorization, rate limiting, and resource quotas at a minimum. You can also turn on input validation in newer API gateways. Yes, you can use serverless apps with API gateways!³⁷
- Ensure isolated function parameters (perform AuthN and AuthZ and input validation to and from any serverless app; do not skip good security hygiene because you're behind the firewall).³⁷
- Don't do too much in one serverless app. Such apps shouldn't do 27 things and run almost all the time, and not only because that would be extremely expensive compared to traditional server hosting. They should do one thing and then disappear; that is the purpose of a serverless app. If you need to do many things, make either many serverless apps or a regular web app or API.³⁷
- * Apply function-level access control (applies to APIs as well) by validating every single request for access. This may seem tedious, but attackers are looking for errors at this level.¹³²
- * Create function timeouts. Each function should have a maximum amount of time it should run (just like it should have a maximum response size). Set a timeout to keep transactions that run longer than the timeout length, as that might indicate an attack is in progress. Issue an alert.¹³³
- Apply network security architecture to the way you roll out your serverless apps. By default, AWS lambda functions run in an isolated Virtual Private Cloud (VPC) owned by the lambda service, but if using a different cloud, put them in an Access Control List (ACL). They should also be monitored, logged, and alerted on; should have firewalls protecting them (subnets); should restrict egress if that's not needed, etc.—just like the rest of your apps.¹³⁴

IoT Security Best Practices

IoT stands for *Internet of Things*, which means physical devices that are connected to the internet, such as a “smart” refrigerator, coffee maker, or watch. Traditionally, the privacy and security on such devices have been . . . not great. These devices sometimes control extremely personal and important systems, such as the locks on your house or a baby monitoring system, meaning security should have the exact same high-priority status it does on any other type of system.

As the joke goes:

“The S in IoT stands for security.”

“But there is no ‘S’ . . .”

“Exactly!”

In this section, let’s cover ways we can improve this!

Best practices for those creating IoT devices:

- Physically secure your IoT devices. If people can’t get into them or at them physically, that can be half the battle. This could mean adding a locking door on your smart aquarium or other physical security controls for your users to take advantage of.
- Secure the network that your devices connect to. Often, IoT devices are connected to home networks, which are left unsecured, with default passwords (or no password at all!), no updates, and terrible configuration choices.¹³⁵ Make securing the device on a home user’s network as simple as possible.
- Ensure that you have a plan to update the software and firmware of your devices regularly. If you are creating consumer devices, set them to auto-update if possible. Assume the worst of your users (sorry, folks!) and try to “force” security on them at every opportunity, including setting secure user defaults.¹³⁵
- Ensure that the mobile app (or anything else it connects to) has good application security. Often, the mobile apps that accompany IoT devices are . . . lacking.¹³⁵
- Deploy your IoT devices on a separate network or segment your network, if possible. Encourage your users to do the same. Imagine yourself creating a moat around your castle to keep badness both in and out of the rest of your network.¹³⁵
- Create a plan to track and monitor your IoT devices on your work networks. Don’t let just *anyone* connect *anything* to your precious network. Instruct your users to do the same.¹³⁶
- Provide multifactor authentication to end users, if at all possible. If it’s offered to you as an end user, enable it.¹³⁶

- If possible, install endpoint protection on the IoT device.¹³⁷
- Use an IoT gateway, similar to an API gateway, to add an additional layer of security. Enable rate limiting, resource quotas, and throttling. Add features to do this for your users, and ideally, also create a setup wizard to make it as easy as possible.
- If you are the end user of the IoT device rather than the person creating it, make sure you lock it down. Turn on every possible security feature. Read the privacy statement and decide if you feel it's acceptable for you and your organization before purchasing, activating, and connecting it to your network. Follow all the same best practices you would for any other device that you are connecting to your office network.

If you are the maker of an IoT device, such as a toaster, think carefully about whether you truly need to turn it into an IoT device. Are you adding value? And if so, is it worth the extra risk? What data do you intend to collect? You will need to protect the data in transit and at rest. Do you have a plan for that? By turning a regular item into an IoT item, you are adding a lot of scope to your product, and you should ensure that it's worth the extra cost. Please don't cut corners or assume IoT doesn't need the same strict precautions that any digital product requires; your customers are counting on you.

We've reached the end of this chapter; give yourself a pat on the back for diving deep into technology-specific security! In this chapter, we've learned about securing APIs, mobile apps, web sockets, serverless, and IoT, which was quite a lot of information. As developers, know that you hold the power to build amazing applications, and with that great power also comes great responsibility. Try your best at the following questions to test your memory and learning from this chapter.

Chapter Exercises

You can find the answers to these questions in Appendix B, "Answer Keys."

1. Name the top threat to APIs on the internet.
2. Can API gateways be used for more than just APIs?
3. Do we still need to apply regular secure coding practices to IoT devices?
4. We should include raw, system-generated error information in our error messages that we show to users. True or false?
5. Do you need to test your APIs as much as your web apps?
6. Why do we need to decommission old, unused versions of our APIs?
7. Why would we want to obfuscate or minify our mobile app code? Why don't we do that with APIs or web apps?

8. Name one way that protecting sensitive information on a phone is different from a regular web app.
9. Why do we only ask for the permissions we need on a mobile app (or any other kind of app)?
10. On Android, why would we want to manually copy over memory, even though we already erased that data?
11. Why would we *not* want our mobile app data to be part of a phone backup?
12. Is IOS memory-safe?
13. Is a web socket one-way or two-way communication?
14. Do web sockets use SSL (Secure Sockets Layer)?
15. Why should we avoid tunneling in web sockets?
16. Why would we want a timeout on our serverless app?
17. Why would we want to apply function-level access control on serverless apps or APIs?
18. Why might we want to install endpoint security products on our IoT devices?
19. Why might we want to segment our network or put firewalls around IoT devices on our network?

Popular Programming Languages

In this chapter, we will dive into securing popular programming languages, including best practices and “gotchas” to avoid. Although programming languages are not secure or insecure by default, the way we use them can *create* vulnerabilities or other security problems. These are implementation issues or bugs (as opposed to flaws or design issues). There are also some parts of programming languages, such as certain functions, lack of memory safety, serialization, etc., that cause high risk, which we should avoid or use with caution. We will go over the important parts for the following languages: JavaScript, HTML/CSS, Python, SQL, Node.js (not a language, but a JavaScript runtime environment), Java, TypeScript, C#, PHP, and C/C++.

For every one of the following programming languages, apply everything you have already learned in this book *plus* the items for the language(s) you are working with.

JavaScript

JavaScript is the world’s most popular programming language; developers *love* JavaScript.¹³⁸ It’s a powerful scripting language that allows developers to create interactive and dynamic content on websites.¹³⁹ It is often used in addition to HTML and CSS to enhance the user experience and add functionality to web

applications. TypeScript, which we will look at later in this chapter, is transpiled (a form of compilation) *into* JavaScript.¹⁴⁰

JavaScript can respond to user actions in real time, online, which was a *game changer* when it came out in 1995.¹⁴¹ It turned static web pages into dynamic applications. Although JavaScript can be run on both the front end and server side, it's used more often for front ends.

On the client side, JavaScript runs directly in the user's browser and allows for dynamic updates without needing to reload the entire page (saving bandwidth). It also provides access to the Document Object Model (DOM), which represents the structure and content of a web page and allows JavaScript to interact with it. On the server side, JavaScript can be executed with the help of runtime environments like Node.js, allowing developers to build scalable and high-performance applications.¹³⁹

JavaScript has its own ecosystem of libraries and frameworks (React, Vue, Angular, etc.) that simplify common tasks and enable developers to build complex applications more efficiently. Several of them provide fantastic security features!

JavaScript best practices:

- Use a JavaScript framework that automatically output encodes everything for you, so you don't have to. Examples include React, Angular, and Vue.js. Be careful to check if there are dangerous functions within these frameworks so you can avoid them, such as React's `dangerouslySetInnerHTML`, Angular's `bypassSecurityTrustAs*` functions, or Vue.js' `v-html` directive.¹⁴⁴
- Perform input validation, outputting encoding, and use the CSP (Content Security Policy) security header to avoid cross-site scripting (XSS).
- Avoid inline scripting. As tempting as it may be to include JavaScript directly within your HTML, it creates the possibility for XSS. It's best to keep your JavaScript in separate external files to maintain that extra layer of security.¹⁴² You can define your scripts in the CSP header. This is similar to inline SQL when we combine user input and then feed it directly to the database to execute. Instead, use *quoted data variables*, such as `src=" . . . "`.¹⁴³
- Always treat user input as something you can display but not use for logic within your application (do not use it to make business decisions). If we all did this, all the time, XSS would disappear. Try to apply this rule or guideline as much as possible, and whenever you cannot follow it (for business reasons, not for personal preference), proceed with caution. Remember: user input is anything the user *could* have changed, not just information from text boxes or intended user input.
- Use `textContent` or `createElement` instead of `innerHTML` or `outerHTML` to prevent XSS.¹⁴³

- Use `encodeURIComponent` and `decodeURIComponent` instead of `escape()` or `unescape()`.¹⁴³
- Use `createElement` and `appendChild` instead of `document.write()`.¹⁴³
- Use native alternatives like `he` or `lodash` instead of `unescapeHTML()`.¹⁴³
- Use `encodeURIComponent()` and `decodeURIComponent()` instead of `encodeURI()` and `decodeURI()` for dissecting or validating a URL.¹⁴³
- URL encode data before passing it to the CSS `url()` method.¹⁴³
- When in the execution context (DOM), URL escape, and *then* JavaScript escape. Yes, you must do it twice, if you are going to put user-supplied data into a URL attribute.¹⁴³
- When in the execution context (between the `<script>` tags), set your string or text content using the `textContent` assignment property to take away its superpowers (ensure that it is not interpreted). For example: `yourElement.textContent = TextFromTheUser;`¹⁴³
- Specify that text is text if you are putting anything from the user into a data element for your JavaScript or CSS. For example, do not use `innerHTML`, instead use `innerText` or `textContent`. Always be specific that it's text; don't let the DOM decide, because sometimes it gets it wrong.¹⁴³
- When setting `element.setAttribute`, only use safe, static attributes (`align`, `alink`, `alt`, `bgcolor`, `border`, `cellpadding`, `cellspacing`, `class`, `color`, `cols`, `colspan`, `coords`, `dir`, `face`, `height`, `hspace`, `ismap`, `lang`, `marginheight`, `marginwidth`, `multiple`, `nohref`, `noresize`, `noshade`, `nowrap`, `ref`, `rel`, `rev`, `rows`, `rowspan`, `scrolling`, `shape`, `span`, `summary`, `tabindex`, `title`, `usemap`, `valign`, `value`, `vlink`, `vspace`, `width`). Do not use dynamic, unsafe attributes (such as `onclick` or `onblur`).¹⁴³
- If doing inline JavaScript (which you should not do, as previously discussed), and using JSON, verify that the `Content-Type` header is `application/json` and not `text/html` to prevent XSS.¹⁴³
- Variables (user data) should only be placed inside CSS property values, not into contexts. `<style> selector { property : $varUnsafe; } </style>`.¹⁴⁴
- JavaScript code should be executed in "strict mode".¹⁴⁵
- Use the `hpp` module within Express to avoid HTTP Parameter Pollution (HPP) vulnerability (where an attacker sends multiple parameters with the same name for `req.query` and/or `req.body`, but their values are malicious).¹⁷³

- Consider using Dom Purify (<https://github.com/cure53/DOMPurify>), which is recommended by several sites to help protect against DOM-based XSS.¹⁴⁶
- Load external resources (including scripts!) securely by using the subresource integrity check when including them in your code. Even if it's from a content delivery network (CDN), check it. Subresource Integrity (SRI) checking is a feature built into modern web browsers that uses a cryptographic hash to verify the integrity of an external script.¹⁴⁸
- Minify and bundle your scripts and add a nonce to secure them further, using Webpack or something similar.¹⁴⁸
- When performing transactions in your apps (for instance, a purchase), pass a token back and forth *outside the cookie* to avoid cross-site request forgery (CSRF). This can be in a hidden field; it doesn't need to be well-hidden to work properly as a defense. This is to protect against phishing attacks where a user clicks a link that includes the initialization of a transaction, assuming the user is logged in to the site that is the focus of the attack (Aliexpress.com or Amazon.com, for example). Even if a user is logged in when they click the malicious link, it wouldn't include the anti-CSRF token, thus stopping the attack in its tracks.
- Use a linter and static analysis (SAST) tool to check your JavaScript for potential vulnerabilities and fix them.
- Audit dependencies using a package manager like yarn or npm.
- Stay up-to-date with your packages. Update them regularly.
- Be extra cautious when using package managers, and verify package names to avoid typo-squatting malicious packages.
- Use `let` instead of `var`. Let declarations are scoped to blocks and functions.
- Use `const` if your variable should not be updated.
- Use strict equality (`===` and `!==`) whenever you can, rather than `==`.

Things to avoid in JavaScript:

- Functions to avoid: `eval()`, `innerHTML`, `outerHTML`, `Function()`, `escape()`, `unescape()`, `document.write()`, `unescapeHTML()`, `decodeURI()`, and `encodeURI()`.¹⁴³
- Avoid using the `with()` statement.¹⁴³
- Avoid function constructors with a string argument.¹⁴³
- Do *not* concatenate untrusted input in JavaScript to create DOM elements or use `document.write()` on dynamically generated content.¹⁹⁷ Instead, do one of the following:

- Use `createElement()` and then assign a property value such as `node.textContent` or `node.innerText`.
- Use `document.createTextNode()` and append it to the appropriate DOM location.
- Use `element.SetAttribute()` *or* `element[attribute]`.
- Use `setTimeout()` and `setInterval()` with caution; validate user input very carefully before considering putting it into one of these functions. Or better yet, don't put user-supplied input into them.¹⁴³
- Avoid JavaScript methods that take code as a string type, such as `setAttribute()`, `setTimeout`, `setInterval`, and `new Function`. Instead, set the attribute directly.¹⁴³
- Do not put untrusted data into HTML rendering methods, as they might be interpreted (XSS is possible). Examples include `innerHTML`, `outerHTML`, and `document.write`, `document.writeln`, etc.
- As JavaScript is a prototyped-based language, you need to avoid prototype injection (I know, another type of injection).¹⁴⁷ You can do this by (1) freezing the default object that everything else inherits from, `Object.freeze(Object.prototype)`; (2) explicitly creating objects that do not use prototyping `Object.create(null)`; (3) using `Map` instead of `Object`; or (4) avoiding recursive merge functions altogether.
- Do not set the same value for `document.domain` to share information between two pages; this ruins the protections provided by the same-origin policy. Instead, use `window.postMessage()`.¹⁴⁷ Please don't set `window.postMessage()` to `*`; make sure you check the origin and validate whatever is in the message before you use it.
- Do not validate your user input *only* on the front end; it must also be done on the backend. Although this applies to every language, the front end is written mostly in JavaScript, and thus it's worth mentioning again in this section.
- Avoid using global variables whenever possible, and instead use private variables, which are less likely to be accessed by external scripts.¹⁴⁹

NOTE With the help of a web proxy such as Burp Suite or Zap, anyone can circumvent the front-end JavaScript of a web application and send requests directly to the web server/backend. As a result, validation, sanitization, or escaping on the front end is never enough from a security perspective. The final decision must always be made on your web server/backend.

In summary, JavaScript is extremely powerful and is part of the front end of almost every website on the planet, and that means you need to be very careful with it. Up next, HTML and CSS, “the looks” of the web front-end.

HTML/CSS

Before you all get up in arms that “HTML is *not* a programming language, Tanya!” I know. HTML (Hypertext Markup Language) is a *markup* language, text that tells a web browser how to display the text, images, and other forms of multimedia on a web page.¹⁵⁰ But it can still present security risks, so we need to take a deeper look.

In this section, we will also cover Cascading Style Sheets (CSS), which describe the presentation of either HTML or XML. You might think that CSS doesn’t present any security risks, but unfortunately, it can! Whether it’s authors revealing sensitive information (different roles in the application or which fields contain secrets), using CSS from untrusted sources, or poor access control, attackers are aware that CSS can be used for reconnaissance or, even worse, when attacking web apps.¹⁵¹ Note: CSS presents only a very small security risk when compared to any programming language, and HTML presents a low risk.

HTML and CSS best practices:

- We must check that the data we receive (the source) is safe to use before we use it (sink) to prevent XSS. As such, HTML encoding¹⁴⁴ is required for anything that was user-supplied or potentially modified by a user (source). Validate the input, and reject it if invalid. If you must accept special characters that are potentially dangerous, validate them and then sanitize your data before using them (sink).
- It’s critical to use quotation marks such as " or ' to surround your variables, to remove their power/denote that they are text, not code.¹⁴⁴ Quoting the value makes it difficult to change the context a variable operates in, which helps prevent XSS. Quoting also significantly reduces the character set that you need to encode, making your application more reliable and the encoding easier to implement. Encoding in multiple contexts is already more complicated than it needs to be; let’s simplify whenever possible.
- Create different style sheets (CSS) for different roles within your application, and then only allow those roles to access those CSS files.¹⁵¹ For every page that tries to load a CSS file, validate first that the role of that user within the system has the right to see that CSS file (and page, data record, etc.). We always check access before we grant it.
- As an alternative to the previous item, you can remove all identifying information from your style sheets so that an attacker would not be able to glean any information about roles or other internal information by

reading them. Unfortunately, this reduces readability for other programmers as well, so this may be a less attractive option.¹⁵¹

- If you know that your CSS contains sensitive information, consider CSS obfuscation¹⁵² to hide potentially sensitive information.
- Exercise caution when putting anything into a hidden field; it's like a sign to malicious actors saying, "Look here!"¹⁵³
- Use all applicable security headers (such as CSP). They *are* HTML, after all.
- Be extra careful with the following elements and attributes, which can be attacked with injection: event handlers, autofocus/on focus, form, and formaction.¹⁵⁴

WARNING Unlike backend code that sits on a web server, anyone with a web browser can select "View -> Source" and see your JavaScript code. As a result, they can see your comments, your logic, and any security measures you have implemented! Proceed with caution.

Things to avoid in HTML and CSS:

- Do not use CSS from an untrusted source. As an alternative, you can copy it, review it carefully, and use the parts you want to keep, *or* your org can choose to accept the risk of third-party CSS if you feel that you can trust where it came from.¹⁵¹
- Don't allow your users to customize your CSS as, of course, someone may choose to attack it (inject their own code, generally JavaScript).¹⁵²
- Don't put user input into your CSS. Or HTML. Or any other code that you write. User input should only be data, never a part of your code.
- Avoid putting anything important or sensitive into comments, as those are clearly visible to any attacker via the browser menu option "View -> Source".¹⁵³

HTML5, Specifically

HTML 5 is the latest and, theoretically, the last version of HTML.¹⁵⁵ It has new features meant to improve the way we create web applications, including geolocation (location of your user), drag and drop (drag and drop elements to different web pages), local storage (stores large amounts of data in the browser), application cache (supports offline browsing), web workers, Server-Sent Events (SSR), and more. As you can imagine, these new features present new security risks.

Special thanks to the people from the OWASP HTML5 Security Cheat Sheet project team for this section!

Security concerns:

- Web storage and online storage (local cache): do not store sensitive data here; use a secure cookie instead.¹⁵⁶
- When using web messaging (not web sockets, which is covered in Chapter 5):¹⁵⁶
 - Explicitly state the expected origin as the second argument to `postMessage`; do not use `*`.
 - When receiving messages, check the origin and validate all input (data attribute).
 - Do not use any data sent back and forth to make business decisions; it's easily changeable by the user (P.S. don't save it into the DOM, either).
 - As previously discussed, avoid using functions like `innerHTML` to assign data values, and instead use the safer `textContent`.
 - If you *must* use a user-supplied script or other external content in this context, proceed with extreme caution, read up on it, and then have it thoroughly tested.
- Caution with cross-origin resource sharing (CORS):¹⁵⁶
 - If you're going to use `XMLHttpRequest.open`, validate against an approved list very carefully. This is dangerous territory.
 - For URLs that you are sharing via `Access-Control-Allow-Origin`, ensure that there's no sensitive data in them. Also, don't use it across your entire site; use it only when you *need* to. Think before you share.¹⁵⁶
 - If you are making a `prefetch` request, that's great, but keep in mind that not everyone will do that, and they will still try to use `GET` and `POST`. Ensure that you have access control on *all* HTTP verbs/methods.¹⁵⁶
 - Do not accept any requests sent over HTTP. It's HTTPS only, all the time.¹⁵⁶
 - Check more than just the `origin` header value for access control decisions. Check the session cookie as well, and the anti-CSRF token, if applicable.¹⁵⁶
- For server-sent events, check the `event.origin` attribute to ensure that it's coming from where you think it is.¹⁵⁶
- When using local storage (AKA offline store and web storage):¹⁵⁶
 - Any user with local privileges on the machine can access this storage, so don't store anything sensitive there.

- Use `SessionStorage` if you only need the data for that session and `LocalStorage` if you need it to persist. Have the data disappear by default (`SessionStorage`) whenever possible.
- Do not trust any data that you have stored in either of these storage options, as it could have been compromised if your app has XSS anywhere in it (it's accessible to JavaScript, and you cannot block it as you can for cookies). That is, unless you encrypted the data before you stored it and then decrypt it before you use it, or perform some other function to ensure both its secrecy and integrity. If you cannot perform encryption or use some sort of system that will encrypt it or ensure its integrity for you, you must validate and escape/sanitize it before use.
- Please don't store your session ID in either of the previously mentioned storage options; that's what secure cookies are for.
- Avoid hosting multiple applications on the same origin: all of them would share the same `localStorage` object. Use different subdomains instead.
- Avoid the "tabnapping" vulnerability (which is no longer considered critical thanks to newer browsers addressing it by default) by using `rel="noreferrer"` or `rel="noopener"` for HTML links and adding them to the `window.open` in your JavaScript. To be extra certain, you can add the HTTP response header `Referrer-Policy: no-referrer` to all HTTP responses. This is mostly backward compatibility at this point.¹⁵⁶
- If you must load untrusted content, use the `sandbox` attribute of your `iFrame` (load it in an `iFrame`). There are more settings you can use to lock it down further. Avoid loading untrusted content if possible.¹⁵⁶

Honestly, when I started writing this part of the chapter, I thought it would be a breeze; I had no idea HTML and CSS could pose so many risks to web applications! I hope this part of the chapter was helpful and didn't scare your pants off.

Python

When it comes to writing great Python code, it's not just about functionality and performance; it's also about protecting your application and your users from potential vulnerabilities and attacks. With Python being extremely popular, it's even more important that we cover securing Python in this section of the book.

SURPRISE QUIZ FOR TANYA

When I did my job interview to work at Semgrep, they made me do a coding exercise, in Python. It was a surprise: I did not know I was going to an interview; I thought it was supposed to be a conversation about how the “secret sauce” of their product worked. I had never worked in or programmed in Python at all. Plus, the exam had recursion and matrix math! I have no idea how I passed, but apparently I did, and the rest is history!

TIP If you find a security bug in PyPi, CPython, or Pip, report it to `security@python.org`.¹⁵⁷ Only do this if you are *sure* you have found a vulnerability; they are busy folks. P.S. Not only can you report bugs in any framework or language, but you should!

What to do:

- Use version 3 of Python, not 2, which is no longer supported.¹⁶¹
- Update your Python environment often. This goes for all frameworks and languages; stay up to date as best you can.
- Use the virtual environment and Pip Freeze to ensure that the modules you ship are the ones you think you are shipping.¹⁵⁹ You do not want to have your local modules interfering with or changing your project modules. Of course, validate that the modules you are using are not known to contain vulnerabilities (if possible), that they are relatively up to date, and that *if there* is a vulnerability in one of them, it is not reachable from within your code.
- Always use either *absolute import* or *explicit relative imports* to ensure that you are getting the correct package.¹⁵⁹
- When taking user input for use as a string in your code, use the `Template` class from the `string` module, rather than other functions, for string manipulation.¹⁵⁹ Avoid using f-strings and `str.format()` for handling user input (or anything that could have been manipulated by the user).
- Remember that the dangers of deserialization do not only apply to Java; Python can also fall prey to malicious actors sending dangerous serialized objects.
- Use tools like `pipenv check` and `safety` to check your dependencies for potential vulnerabilities and manage your dependencies.¹⁶⁰
- Scan your code with Bandit (a free and open-source tool) to look for vulnerabilities. This one is Python-specific.¹⁶²

- Keep your version of “certifi” up to date. It’s used by the requests library to verify that certificates are on the up and up.¹⁶¹
- Store secrets either in environment variables (various tutorials available online) or a secret management tool.¹⁵⁹
- Consider using a linter on your code (in addition to a static application security testing tool, SAST), such as pep8, pylint, or flake8. It increases code quality and readability, which helps with security and maintainability.¹⁶³
- Ensure that you are using the latest version of Python, not the pre-loaded default POSIX version,¹⁶³ as they are often out of date.

TIP You might have noticed that I suggested Pip Freeze and then said not to pin your libraries. How can you both freeze *and* keep updating libraries? Whenever you have a chance to do a code update, you want to update as many libraries as you can, to avoid technical debt. If you have the libraries pinned, they won’t do that. But when you move from environment to environment (dev -> QA -> staging), you do not want versions of your code changing, as your testing will be inaccurate. Once you are ready to go beyond dev, you freeze, but before that, you update, update, update (especially libraries!).

What *not* to do:

- In Python 2, avoid `input()`; instead, use `raw_input()`. For Python 3, `input()` is okay to use.¹⁵⁸ P.S. Don’t use Python 2 anymore.
- When using PyYAML to load YAML, the default loader, `yaml.load()`, is unsafe. Don’t use it.¹⁵⁸
- Do not use subprocesses with `shell=True` and then feed Python code into it from the user. It’s a very bad idea.
- When importing packages, never use the “implicit” path; always use “explicit” (it’s no longer allowed in Python 3) to avoid accidentally getting the wrong package.¹⁵⁹
- When using libraries, you should not pin the versions into your `requirements.txt`, because it might install an outdated and vulnerable version of the module. Don’t pin; keep updating.¹⁵⁹
- Avoid using the following functions with anything that was user-provided or user-tamperable: `eval()`, `exec()`, `compile()`, or any other function that performs system actions or executes code based upon the parameters you feed it.¹⁶⁰

Up next, SQL!

SQL

Gather around the (imaginary) campfire for some thrilling and scary information about SQL! Of course, we are going to cover the infamous SQL injection attack in this section of the book (and how to avoid it), but also other issues you may not have heard of before. I'm also going to preach at you, again, that you need to harden and patch your SQL server(s) regularly if you want to keep your data safe. And, of course, you want to keep all that precious data as safe as can be, so let's go learn!

Security tips for SQL language:

- Use an ORM (Object Relational Mapping) framework on top of your SQL, to help you avoid SQL Injection and more!
- Encrypt your data at rest and transit to and from your database.¹⁶⁴
- Use stored procedures when allowing access from an application or another system, rather than direct querying of the data.

Security tips for hardening an SQL database and server:

- Harden your SQL server and the underlying operating system it lives on.¹⁶⁵ Scan it with a database and/or vulnerability assessment (VA) scanner, and follow all the advice—fix everything. Then get the hardening guide from the vendor that created it, and follow all of that advice, too.¹⁶⁴
- Keep your database server software up to date. Patch and scan it regularly.
- Always use a service account in your application to access the database, never a person's account.¹⁶⁴
- Do not give administrative access to your DB unless that person is the database administrator. Developers generally do not need to administer the database.¹⁶⁴
- Do not use accounts with Database Owner (DBO) privileges for your app. It's highly unlikely that you *need* DBO and more likely that you would *prefer* to have it. But using read-only or CRUD is more likely what you require. Use the least amount of privileges you need for app accounts to reduce the damage if things go wrong.¹⁶⁴
- Classify (figure out if it's sensitive or not, and if so, how sensitive) and label your data (literally mark it somewhere with what classification level it is, even if it's public/not sensitive).¹⁶⁴ This action makes it simple for everyone to know exactly how to treat the data. You can do this at the database level, table level, and even field level.
- Protect physical access to the server. No one should be able to physically get at your servers unless they are on your Ops team or work at your cloud provider.

- Ensure that logging is turned on so that you have something to investigate if something goes wrong.¹⁶⁴ This means access logs (who accessed the database and when), query logs (what queries were executed), and, if possible, also meta-logs from your cloud provider (for example, they will tell you when a new security policy was added to the DB).
- Ensure that you have a proper backup strategy for your logs and that only people who should have access do. Your logs are very sensitive.
- Turn on monitoring, so you know if your database goes down.¹⁶⁴ This may or may not be a security issue (database crash/down).
- Enable multifactor authentication (MFA) for all database users.
- Install File Integrity Monitoring (FIM) and Application Control tooling on your server, to protect against malware.
- For highly sensitive applications and workloads in the cloud, consider “Confidential Compute” virtual machines, a special class with extra protection against side-channel and other attacks.¹⁶⁵
- Have an expansive and well-thought-out backup plan, and practice your rollbacks. Ideally your backups would be geographically distributed (not all in the same building, city, or even country), with disconnection of some kind in case of ransomware (you would not believe how many people leave their backup drive connected to their machine, and it gets encrypted right along with everything else on your network during the attack).
- Give your local administrator account a username that is not “admin” or “administrator.” Choose something else. Anything else.¹⁶⁵
- Use strong, long, unique passwords to access your database, ideally created by a password manager.¹⁶⁵
- Use a different account for backups than you use for administration.¹⁶⁵
- Do not install anything else on your database server. It should only do this one, very important, thing.¹⁶⁵
- Following networking security best practices on your server, such as putting it behind a firewall and locking down your ports.¹⁶⁵
- Set up a system so that when an employee leaves, you know to delete their accounts. This goes for every system where you work, from email to HR accounts.¹⁶⁶
- Audit access to your database server. Are people connecting who shouldn’t? Why? Find out.¹⁶⁶
- Disable the SQL Server Browser Service, as it gives important information away to unauthenticated users.¹⁶⁵

Security concerns for SQL:

- SQL injection is the best-known vulnerability that SQL faces. It happens when a malicious actor tricks the application into appending their code onto some of the application's code legitimate code, turning it into an attack, and then sends it to the SQL server as a command to be executed (as opposed to using a stored procedure). This is generally done by entering the attack code into a field that *should* be treated as data, such as a search field.
- Avoid SQL injection by using stored procedures (instead of sending commands directly to be executed) and using input validation. Also, test your app for this vulnerability with a penetration test or something like SQLMap, use a SAST to check your code, and then apply least privilege to your database accounts to reduce damage if you end up getting attacked successfully despite your best efforts. SQL injection is *terrifying*, and it is absolutely worth doing *all* of these steps to avoid it.¹⁶⁴
- Do not put production data into nonproduction environments. This is a huge security risk and most certainly *not* a best practice (despite being extremely common). There are many products and plugins that can create test data for you, such as the Bogus NuGet package for .NET/C#. If you can't get your hands on usable test data (data that is *not* a copy of production), take your copy of production and mask it (switch the fields or all the records around so that it's impossible to see which parts belong to whom). There are tools for this; do not do it yourself, as the data may be easily de-randomized. This will help ensure that private customer data remains that way: private.¹⁶⁴
- Ensure that the accounts you use for nonproduction environments to access the database are different than the accounts you use for production. The same service account should not be able to edit dev and prod data.
- Do not use the same secret (the password in the connection string specifically) to connect to prod as any other lower environment.
- Any risk that applies to a server generally also applies to SQL servers. Protect your database server software the same way you would any server (from malware, theft, physical attacks, power surges, overheating, etc.).
- Review all SQL code that calls `EXECUTE`, `EXEC`, `sp_executesql`, or any catalogue-extended stored procedures: `xp_*`.¹⁶⁵
- Suppress server errors from being shown to users; always use a custom error message (that does not give system information, especially not syntax).¹⁶⁵
- When creating your approved lists for input validation, do not include the following characters and character combinations as "allowed": `;` , `"` , `--` , `/` , `*` , `*` , `/` .¹⁶⁵

Although there is more you can do in this area from a networking/operational perspective, this isn't a networking book, and I didn't want to dive too deep into suggestions that you, as a software developer, likely do not have access to. That said, please do as much as you can to protect your data; your users will thank you for it.

Node.js

Welcome to the world of Node.js, where the power of JavaScript meets the server side of our applications. As developers, we likely love the speed, scalability, and flexibility that Node.js offers. But along with these advantages comes the responsibility to ensure the safety and reliability of the applications we build with it.

Node.js is an open-source, server-side JavaScript runtime environment built on the Chrome V8 JavaScript engine. With it, we can run JavaScript code *outside the browser*, making it possible to build scalable and high-performance applications that do not run on the internet (not just web apps, like JavaScript).¹⁶⁷

One of the key features of Node.js is its event-driven architecture and non-blocking I/O model. This means that Node.js can handle multiple concurrent connections efficiently, making it ideal for building real-time applications such as chat systems, streaming platforms, or collaborative tools. Node.js is not generally considered suitable for building heavy computing, such as scientific computing (giant calculations) or simulations (such as a flight simulator).¹⁶⁷

Node.js has a huge community that creates and manages its vast ecosystem of modules and packages available through the Node Package Manager (NPM). Node.js is a popular choice for building web servers and APIs thanks to its ability to handle high traffic loads and asynchronous operations and the fact that then developers only need to learn one language (as opposed to JavaScript for the front end and another for the backend).¹⁶⁷

What are you waiting for? Let's go!

Best practices for Node.js:

- Don't run Node.js as root in prod. If you're in a workshop learning, sure. If you're creating something that people will depend on, don't run anything as root, as a general rule.¹⁶⁸
- Use NPM audit or the Snyk NPM CLI tool to check your dependencies and keep them up to date. They are both free.¹⁶⁸ There are also lots of paid tools that can help.
- Use one of the Node.JS security linters (in addition to a SAST, not instead of), such as ESLint, JSLint, JSHint, quick-lint-js, or StandardJS.¹⁶⁹
- Avoid blocking the event loop, which can be used as a denial of service against your app. Keep your threads open and available.¹⁷¹

- Write and test your callbacks to ensure that they are of low complexity (using a consistent number of steps) and execute quickly.
- For more complex tasks, bound the inputs and reject long inputs. Bound inputs execute against the timeframe of your worst-case input so the thread won't block. Ensure that your worst-case input timeframe is short to avoid blocking the thread. This might take some testing and guesswork.
- Avoid using synchronous APIs from the encryption, compression, file system, or child process modules. They take a long time and can block your event loop, so run them asynchronously.
- Prevent `npm` from executing arbitrary scripts by using `--ignore-scripts` and/or disabling it globally with `npm config set ignore-scripts true`.¹⁷²
- Use `npm ci` instead of `npm install` if you are using a lockfile to lock your dependency versions. This causes an error if there are inconsistencies between it and the `package.json` file (instead of silently ignoring the lockfile in favor of `package.json`).¹⁷²
- When using `npm ci`, add the `-production` flag to ensure that you only install the minimal requirements needed for production (it will ignore test utils and other nonproduction items).
- Use an `async-await`, rather than a callback, or a reputable promise library (as a second choice) for managing asynchronous errors.¹⁷¹
- Use the built-in error object rather than creating your own or using the string error handling.¹⁷¹
- Use a process management tool to restart when an unknown error occurs. *Pm2* and *forever* are examples of Node.js process management tools.¹⁷¹
- Use helpful NPM-specific input validation tools like *DOMPurify* and form validation tools like *express-validator* or *XSS-filters*.¹⁷¹
- For sensitive operations, use user interaction-based protection.¹⁷¹
- Use custom request headers instead of the standard ones.¹⁷¹
- Use flat Promise chains¹⁷³ to ensure that your errors and results get back to where you called your `async` callback, even if it's recursed 10-plus times.
- Use input validation modules, such as *validator* and *express-mongo-sanitize*.¹⁷³
- Use the *toobusy-js* module (or something similar) to monitor the event loop for potential DOS attacks or other potentially crash-inducing situations.¹⁷³
- Use modules such as *Express-bouncer*, *express-brute*, and *rate-limiter* to slow down (throttle) login requests if it seems like a brute-force attack might be happening.¹⁷³

- Remove unused API routes (where a user can send a request, and the API will respond, even though you didn't program it to do so). This is especially important in frameworks such as Sails and Feathers, where they "help" you fill in the blanks by generating any "missing" API endpoints for you. This is bad, as you are not in control of those responses; shut it down.¹⁷³
- If your object properties do not need to be editable/deletable/used in `for` loops, set your properties accordingly for *writable*, *configurable*, and *enumerable*.¹⁷³
- Listen to error events when using `EventEmitter` objects; they can be anywhere in the event chain. Your first argument for asynchronous calls should be the `Error` object; otherwise, they aren't handled in the express route.¹⁷³
- If using regular expressions, avoid the ReDOS (regular expression DOS) attack by checking first if something is dangerous before putting it through your regex, using `vuln-regex-detector`.¹⁷³
- Use strict mode, just as we said in the JavaScript section.¹⁷³

Security concerns for Node.js:

- The most prominent attacks on the NPM ecosystem have been supply chain attacks where external NPM libraries were injected with malicious code. Thus, I want to emphasize again how truly vital it is to keep your dependencies up to date, to only use external libraries where they're truly needed, ensure you get your libraries from a trusted source, and to use tools like `npm audit`, the free NPM CLI version of Snyk, or another software composition analysis tool.
- Denial of service (DOS) for HTTP server requests.¹⁷⁰ The way Node is designed for the backend, a request that looks good can still be used to attack the backend if it's split into pieces such that it eats up all your resources (CWE-400). Mitigation:
 - Use a reverse proxy (or Web App Firewall [WAF] or load balancer if not available), and enable all potential security features to protect your app from this kind of attack, especially setting a max response size and anything to do with rate limiting.¹⁷⁰
 - Configure your server timeouts carefully so that you don't have idle threads or (problematically) slowly arriving requests.
 - Limit your number of open sockets (total and per host) so you don't get bogged down.¹⁷⁰
- Limit request size (to avoid DOS) by configuring the "limit" option in the `body-parser` package to only accept small payloads. Also, when using reverse proxies or express middleware, set size limits for certain types of content.¹⁷¹ Just because requests *can* be 5M doesn't mean they *should*.

- Avoid DNS rebinding attacks (CWE-346) by disabling the inspector on SIGUSR1, and do not run the inspector protocol in production. Just like debug, it's not for prod.¹⁷⁰
- Avoid publishing potentially sensitive information from your files and folders in your current directory when you publish to a package by.¹⁷⁰
 - Using `npm publish --dry-run` to list all the files to publish; check them carefully.
 - Using `.gitignore` and `.npmignore` to list which files not to publish.
- Avoid using the following dangerous functions: `eval()` and `child_process.exec()`. Use the following modules with extra care: `fs`, `vm`.¹⁷³

Java

I learned Java in college after learning Pascal, C, and C++. It seemed so *modern* with its memory safety, garbage collection, and fancy virtual machine. It could run *anywhere*! And the apps were *huge*. That course also introduced object-oriented programming (OOP) to me, which was a brand-new and amazing concept for my 20-year-old brain to absorb.

Nowadays, when I'm told an application is written in Java, I assume it's an old monolith. I assume I'm going to find some vulnerabilities and that I can use my old tools, such as DAST (dynamic application security testing), that don't work as well on more modern application architectures. That's not to say that an application written in Java can't be modern, tough, rugged, and secure! With almost half (49%) of developers saying they use Java,¹⁷⁴ this is an incredibly important language that we need to be sure is easy to understand how to code in securely.

Java security best practices:

- Follow secure coding guidelines provided by Oracle, available here: <https://www.oracle.com/java/technologies/javase/seccodeguide.html>.¹⁷⁵
- Also, take a look at the CERT Oracle Secure Coding Standard for Java, by Fred Long.¹⁷⁶
- Keep your Java SDK up to date.¹⁷⁵
- Keep the Java Virtual Machine Secure:¹⁷⁵
 - Configure the JVM security settings to restrict permissions based on the principle of least privilege.
 - Utilize security managers to enforce security policies and prevent unauthorized actions.

- Minimize the number of permission checks you do on your objects, because the more you check, the easier it is to screw up.¹⁷⁵
- Encapsulate your objects. Each interface of a method, class, package, and module should form a coherent set of behaviors and no more. Do *not* create backdoors or extra interfaces so you can grab something more easily later. Build it in, or don't do it at all.¹⁷⁵
- Even though Java is a memory-safe language, it can still have DOS attacks against it. Whenever possible, limit input to something "reasonable," as defined by your product team.¹⁷⁵ Use rate limiting as well, and beware of activities that may use disproportionate resources, such as very large graphic types, zip bombs, etc.
- Ensure that you release resources that you have locked, such as locks, open files, and manually allocated memory.¹⁷⁵
- Check your bounds on integers, even though Java is memory-safe. It's not perfect.¹⁷⁵
- In Java, when an exception happens, you can *handle* it, or you can *propagate* it. Quite often, the place where the exception happens can't handle it properly, so propagation is common. Make this a part of your design; don't "play it by ear".¹⁷⁵
 - *Handling* means catching it, doing something (usually clean up), and then continuing as normal. You have shielded yourself from the exception.
 - *Propagating* means handing the exception back to whoever called your function/code/whatever for *them* to handle. Ideally, then, it's handled wherever it was originally called from.
- Do not allow usage of XML Document Type Definitions (DTDs); if you absolutely must, proceed with extreme caution.¹⁷⁵
- Avoid using BMP files, or restrict their ability to read local International Color Consortium (ICC) files.¹⁷⁵
- Floating-point numbers (floats) in Java can be exploited via the NaN (not a number) or infinite numbers. Use strict input validation if you are going to take a float from an untrusted source and use it within your application.¹⁷⁵
- Make all classes, interfaces, class members, and constructors private or package-private unless there is a business reason to make them public (such as it being part of a public API). Default to being locked down. Document every time you choose "public" and why.¹⁷⁵
- Modules in Java strongly encapsulate (protect) the classes and interfaces inside its packages. Use a module to hide your package contents. Only export packages if you have a business reason, and document it.¹⁷⁵

- Limit exposure of `ClassLoader` instances because they are powerful and contain operations you may not want to share.¹⁷⁵
- Limit the extensibility of classes and methods by declaring them `final`. Decide if they will inherit or be “final.” Do not leave them non-final *and* not inheriting; you need to decide and then code it that way.¹⁷⁵
- If an object is supposed to pass back a value to its caller that it got from an untrusted source, validate it first. Always validate all input before you use it. Example: Class objects returned by `ClassLoaders`.¹⁷⁵
- Although pure Java code is effectively immune to traditional buffer-overflow attacks, native methods are not.¹⁷⁵ To protect your native code, don’t declare it public; instead, declare it private and expose the functionality through a public Java-based wrapper method.
- Whenever possible, use immutable data types, classes, output types/values, etc.¹⁷⁵ If you do need to update or pass the value, do this by creating a safe copy and passing that instead.
- Make public static fields `final`. Also, make them constants.¹⁷⁵
- Do not expose mutable statics or modifiable collections. Expose a read-only copy instead.¹⁷⁵
- Prevent the unauthorized construction of sensitive classes by not exposing them. Define static factory methods instead of public constructors, and perform a `SecurityManager` check.¹⁷⁵
- If a constructor does not complete error-free, do not allow it to be used. Destroy it and start again if there is an exception. Ensure that it is unusable when you handle or propagate whatever happened.¹⁷⁵
- Prevent constructors, `clone`, `readObject`, and `readObjectNoData` from calling methods that can be overridden to avoid an attacker attempting to access `this()` and using it for their own purposes. Ideally, you would label all methods that can be overwritten to make this easier.¹⁷⁵
- Don’t allow non-final classes to be cloned, as this can cause all sorts of issues with two copies (with one not controlled by you). Ideally, you would label all non-final classes to make this easier.¹⁷⁵
- Limit the use of callback methods. If you are going to use them, ensure that you transfer context and handle access control strictly.¹⁷⁵
- Exercise caution and restrict privileges when invoking `java.security.AccessController.doPrivileged`. It is very powerful and lets code exercise its own permissions when performing `SecurityManager`-checked operations.¹⁷⁵
- Do not cache the results of a privileged operation and then pass it to a context that does not have the same permissions. Although this is rather

unusual and specific, you know it's in the Oracle secure coding guideline because someone did it, and there were dire consequences.¹⁷⁵

- SecurityManager checks can be bypassed by several methods in the core libraries of the Java runtime. Why? I have literally no idea why that is a good idea. Instead of checking all the way up the line, it just checks the direct caller's permissions and grants accordingly. This is called being *caller sensitive*. If you thought the restriction you put on the top-level class would be inherited all the way down . . . that might not be true.¹⁷⁵ Here's the list for you to test the permissions on very, very carefully:

- `java.io.ObjectStreamField.getType`
- `java.io.ObjectStreamClass.forClass`
- `java.lang.Class.newInstance` (deprecated)
- `java.lang.Class.getClassLoader`
- `java.lang.Class.getClasses`
- `java.lang.Class.getField(s)`
- `java.lang.Class.getMethod(s)`
- `java.lang.Class.getConstructor(s)`
- `java.lang.Class.getDeclaredClasses`
- `java.lang.Class.getDeclaredField(s)`
- `java.lang.Class.getDeclaredMethod(s)`
- `java.lang.Class.getDeclaredConstructor(s)`
- `java.lang.Class.getDeclaringClass`
- `java.lang.Class.getEnclosingMethod`
- `java.lang.Class.getEnclosingClass`
- `java.lang.Class.getEnclosingConstructor`
- `java.lang.Class.getNestHost`
- `java.lang.Class.getNestMembers`
- `java.lang.ClassLoader.getParent`
- `java.lang.ClassLoader.getPlatformClassLoader`
- `java.lang.ClassLoader.getSystemClassLoader`
- `java.lang.StackWalker.forEach`
- `java.lang.StackWalker.getCallerClass`
- `java.lang.StackWalker.walk`
- `java.lang.foreign.SymbolLookup.loaderLookup`

- `java.lang.invoke.MethodHandleProxies.asInterfaceInstance`
- `java.lang.reflect.Proxy.getInvocationHandler`
- `java.lang.reflect.Proxy.getProxyClass` (deprecated)
- `java.lang.reflect.Proxy.newProxyInstance`
- `java.lang.Thread.getContextClassLoader`
- `javax.sql.rowset.serial.SerialJavaObject.getFields`

Serialization in Java

Deserialization of untrusted objects in Java is dangerous and should be avoided.¹⁷⁵ Here are some tips to handle serialization more safely:

- Don't serialize security-sensitive classes.
- Protect sensitive data during serialization by¹⁷⁵
 - Declaring sensitive fields *transient*
 - Defining the `serialPersistentFields` array field appropriately
 - Implementing `writeObject` and using `ObjectOutputStream.putField` selectively
 - Implementing `writeReplace` to replace the instance with a serial proxy
 - Implementing the `Externalizable` interface
- Design your deserialization to act like a constructor (it *is* building an object for you, after all). Use `ObjectInputStream.readFields` instead of `ObjectInputStream.defaultReadObject`. Perform input validation checks for the `readObject` method. Create copies of deserialized mutable objects before assigning them to internal fields in a `readObject` implementation.¹⁷⁵
- Perform the same security-related checks in a class during both serialization and deserialization.¹⁷⁵
- Verify your permissions for `SecurityManager` carefully as they apply to deserialization.¹⁷⁵
- Use the serialization filtering feature.¹⁷⁵
- Check out the OWASP Deserialization Cheat Sheet; it is *very* thorough.¹⁷⁷

Things to avoid in Java:

- Avoid `Method.invoke()`.¹⁷⁵
- Avoid using caller-sensitive method names in interface classes.¹⁷⁵
- Access to `InvocationHandlers` should not be generally available (restrict it).¹⁷⁵

- Avoid using Java Native Interface (JNI) whenever possible. It is high risk.¹⁷⁵
- Avoid injection when using Java Persistence Query Language by using parameterized queries, as well as input validation.¹⁷⁹
- Avoid XPATH injection by using XPath Variable Resolver.¹⁷⁹
- Avoid using the built-in **Java** Cryptography Architecture (JCA) and **Java** Cryptography Extensions (JCE) classes for encryption; instead, use third-party components for this that are easier to implement securely (because more of the work is done for you).¹⁷⁹

Although there is more to write on the topic of Java, this book can only be so long (so says my publisher). I strongly suggest you check out the SEI CERT Oracle Coding Standard for Java on top of my previous recommendations.¹⁸⁰

TypeScript

TypeScript is a new and trendy language created by Microsoft,¹⁸¹ it is a superset of JavaScript. It introduced static typing and additional features to allow developers to write cleaner, more maintainable code by catching potential errors and providing better tooling support during development. TypeScript code is *transpiled* into plain JavaScript, making it compatible with all JavaScript engines and browsers. It provides advanced features such as classes, interfaces, modules, and generics, making it easier to build large-scale applications. TypeScript's static typing enables developers to detect bugs before runtime, leading to more robust and reliable code. It also supports modern JavaScript features and provides excellent editor support, including code completion, refactoring, and debugging capabilities. TypeScript is widely adopted and supported by major frameworks and libraries, making it an ideal choice for building scalable and maintainable web applications.¹⁸²

Things to do in TypeScript:

- Turn on strict mode (just like in JavaScript).¹⁸²
- Do not use the `any` data type; define your data type. Be specific.¹⁸³ When we leave things unknown or ambiguous, that is where vulnerabilities lie.
- Use `let` instead of `var` to declare variables with a clear scope that can't be sneakily redefined on you elsewhere.¹⁸³
- Declare constants as `const`. This ensures that they aren't accidentally redefined elsewhere.¹⁸³
- Use tuples for fixed-length, typed arrays (this is a properly defined array so that only the exact number of items can be entered; otherwise there is a syntax error).¹⁸³

- Be careful when deciding which properties and methods of your classes are public, private, or protected, especially when sensitive data is involved. Remember: all classes are public in TypeScript.¹⁸³
- Exercise caution when using `DefinitelyTyped` and other special types for packages that do not come with a type definition of their own. Ensure that it matches the output correctly.¹⁸⁴
- In both TypeScripts and JavaScript, use `===` and `!==` rather than `==` or `!=`. The reason for this is that if you are comparing different types, it could lead to ambiguity.¹⁸⁵
- Set default values for your function parameters to initialize them. Otherwise, they are left undefined, and if for some reason you do not set them and then attempt to evaluate them, your code could fall into an unknown state.¹⁸⁵ Also initialize all variables for the same reason.

TIP Declaring constants as `const` applies not only to TypeScript but also to JavaScript and Node.js! Let's make sure our constants aren't accidentally redefined elsewhere in all the languages.

Things not to do in TypeScript:

- Avoid publishing source maps in production, as they can reveal the original TypeScript code.¹⁸⁴

You might have noticed this section is a little light. As TypeScript has some security features built into it (thanks to it being strongly typed), if you apply all you learned about JavaScript and the agnostic coding items that we covered in Part I, plus a secure system development life cycle (SDLC), you should be all set!

C#

When I was a software developer in the government, they had me code in ASP, then VB.net, and I just got into C# before moving on to Microsoft, the center of the .NET universe! By then, I had been working in security for a few years and was more than a little rusty in my programming skills. Although I was quite disappointed to find out that the “visual” drag-and-drop part of Visual Studio had been removed (apparently no one liked it but me?), I was delighted to work with a framework and language designed and maintained with security in mind.

C# is a statically typed language, providing robust type checking, which enables early error detection (during compilation instead of runtime).¹⁸⁶ It is also memory-safe and object-oriented and can be compiled into bytecode.¹⁸⁸

What to do in C#:

- Ensure that you keep your framework and dependencies as up-to-date as possible, and remove all unused dependencies before deployment. It's just extra baggage you don't need. There's a feature in VS Code and Visual Studio that will automatically remove unused dependencies for you.
- Use the Entity Framework (the ORM) when working with SQL Server databases for better security.
- Use AntiXssEncoder or HTML encoding features that are part of the framework for output encoding.¹⁸⁷ This helps with *encoding*.
- Consider using libraries such as AntiXSS to assist in user input sanitization (after you have validated it).¹⁸⁷ This helps with *sanitization*.
- Use ASP.NET identity to manage sessions and identity.¹⁸⁷
- Use the ASP.NET Request Validation feature for input validation *and* sanitization.¹⁸⁷
- Use the ASP.NET MVC built-in support for antiforgery tokens (to avoid CSRF).¹⁸⁷ Do this by adding `@Html.AntiForgeryToken()` to your `_Layout.cshtml` file.
- Use the `SameSite` cookie attribute in the `Startup.cs` file to stop CSRF: `options.Cookie.SameSite = SameSiteMode.Lax;`¹⁸⁷
- Use a strong hashing algorithm such as bcrypt or PBKDF2.¹⁸⁷
- Use either the Regular Expression Attribute or Regular Expression Object Model when creating an allowlist for input validation.¹⁸⁸
- Use the MVC Razor engine because it automatically encodes all inputs, helping you avoid XSS.¹⁸⁸
- URL encode your query strings to avoid XSS using the built-in NuGet libraries: `System.Net.WebUtility.UrlEncode` and `System.Net.WebUtility.UrlDecode`.¹⁸⁸
- For custom error handling, either use `ExceptionHandlerAttribute` (you will need to override the `OnException` method and write your own exceptions) or configure `UseExceptionHandler` to work in the production environment (in lower environments, it will show the regular system message; in production, it will show the custom messages).¹⁸⁸
- Remove system version information from responses.¹⁸⁸

- Remove the `Server` header by adding the line `UseKestrel(c => c.AddServerHeader = false)` in the `CreateWebHostBuilder` method in the `Program.cs` class.
- Remove the `X-Powered-By` header by adding `<remove name="X-Powered-By" />` to the web configuration file (create one if you don't have one). It should go in the `<customHeaders>` section. It can also be removed from the Internet Information Services (ISS) settings if that's easier for you.
- Remove the `X-AspNet-Version` header by adding `<httpRuntime enableVersionHeader="false" />` to the web configuration file.
- Remove the `X-AspNetMvc-Version` header by adding `MvcHandler.DisableMvcResponseHeader = true;` to the first line in `Global.aspx`.
- When authenticating users, consider using the native options of Windows Authentication for Windows apps and using Form Authentication for web apps.¹⁸⁹ Or use industry standards like OAuth and OpenID Connect or third-party authentication services such as X (formerly known as Twitter) and Google, with the help of the `IAuthentication` service.¹⁹⁸
- For authorization to pages and files, consider using the native file authorization (like an access control list) and URL authorization (which URLs which users are allowed to access).¹⁸⁹ You can also use the authorization components, including the `AuthorizeAttribute` and `AllowAnonymousAttribute` attributes, which are in the `Microsoft.AspNetCore.Authorization` namespace.¹⁹³
- Use the Membership Framework Membership provider class to manage your app's users, plus the built-in Login Web controls.¹⁸⁹
- Use the `ILogger` API or TradeSource (in the .NET SDK) for logging or a third-party library such as Log4Net, Nlog, or serilog.^{190, 191}
- Take advantage of the Code Access Security (CAS) feature in .NET, which defines permissions for how your code runs. It can ensure that your code doesn't access system files and other things you would not want it to do. Think of it as applying least privilege to your code; in case your app is compromised, you will have greatly reduced the damage.¹⁹²
- Use the namespace `System.Security.Cryptography` for hashing, digital signatures, secure random number generation, and more.¹⁹²
- Use ASP.NET Identity to manage identity, if not a third-party product or something provided by your cloud provider.¹⁹²
- Use Microsoft's Roslyn analyzer to look for vulnerabilities in your code (for free) or a paid static analysis tool.¹⁹²

- Use the `Microsoft.AspNetCore.DataProtection` packages to manage keys, perform cryptographic operations, and more if you are using .NET Core.¹⁹³
- Use the ASP.NET Core anti-forgery features in ASP.NET Core Data Protection to avoid CSRF unless you are creating a Razor page, which already contains anti-CSRF tokens.¹⁹⁶
- Use Razor, if possible, as it automatically output encodes all output sourced from variables for you.¹⁹⁷
- Use encoders provided to you by your framework via the `System.Text.Encodings.Web` namespace. You can even customize them if you want.¹⁹⁷
- Use `System.Security.Permissions.PrincipalPermission` objects to implement role-based access control (RBAC) within your applications, which is part of the .NET framework.¹⁹⁸
- Use NuGet to keep your packages up to date.
- If you use `XmlTextReader` to parse XML files, you must set the `DtdProcessing` property to `Prohibit` or `Ignore` to avoid the XML External Entities (XXE) vulnerability.¹⁹⁹
- Use `System.Text.RegularExpressions` to create regular expressions (regex) that you use for input validation.²⁰⁰
- Use `System.Security.Cryptography` for cryptographic activities like hashing and encrypting.²⁰⁰

What not to do in C#:

- When using .NET Core, do *not* use `RequireHttpsAttribute` on web APIs that receive sensitive information because API clients may not understand or obey redirects from HTTP to HTTPS. Either ignore (not listen to) HTTP requests or send back a 400 response.¹⁹⁴
- Don't use the Secret Manager tool for prod, as it doesn't encrypt the stored secrets and shouldn't be treated as a trusted store.¹⁹⁵
- Only use `@` in an HTML context, not when attempting to insert untrusted input directly into JavaScript. Razor only handles the HTML context for you.¹⁹⁷
- Don't put untrusted data into the `HtmlString` class, as it is not automatically encoded for you.¹⁹⁷
- .NET compiles code into the Microsoft Intermediate Language (MSIL), which has built-in checks to validate that it is only used in specific memory locations. Do not mess with this and undo this protection by overriding the verification step.¹⁹⁸

- Don't use the `eval()` function with user-supplied values in any language.¹⁹⁸
- Don't write your own security functions instead of using the ones provided by the framework. .NET maintainers are obsessed with security and build it into the framework on purpose. Take advantage!

You may have noticed that quite a bit of “what to do” in #C concerns using the security functionality built into the .NET framework, and that is okay! Use everything provided that gives you value. On top of this, Microsoft provides Visual Studio Code, an excellent IDE that is wildly popular, for free. Use that, too! And while you're at it, add as many security plugins to it as you can; its ecosystem is feature-rich with many choices to help you write even better code than ever before.

PHP

PHP (which is named as a recursive acronym for Hypertext Preprocessor) is a wildly popular, open-source, general-purpose scripting language that works very well for web development (and used to be a common choice for e-commerce sites).²⁰¹ Quite a lot of legacy websites and systems are built in PHP, and often, when I go on an AppSec consulting engagement, if there are PHP apps, I find a lot of vulnerabilities there—not because PHP isn't a good language but because the older the application, the more likely security was not a consideration when it was written.

PHP code is executed on the server (not the browser, like JavaScript), which generates HTML that it sends to the client (browser). This means that end users are not able to see the code you wrote, only the generated code, which creates a level of abstraction and obfuscation.²⁰¹

The folks who maintain PHP created a security manual and share it freely; it is available here: <https://www.php.net/manual/en/security.php>. Warning, though: it's . . . an intense read. Set half a day aside if you plan to read it and fully understand it.

What to do in PHP:

- When installing PHP as a Common Gateway Interface (CGI) binary on your server:²⁰²
 - Use the runtime configuration directives `cgi.force_redirect`, `doc_root`, and `user_dir` to prevent access to your `/secret/doc.html` directory.
 - Use the configuration directive `cgi.force_redirect` to ensure that users cannot access calls to your PHP server directly. All requests then have to go through a redirect that you control.

- Set the PHP script document root with the configuration directive `doc_root` in the configuration file, or set the environment variable `PHP_DOCUMENT_ROOT` to ensure that no scripts run except the ones you plan to run.
- Consider putting the PHP parser binary somewhere outside of the web tree of files as a more secure option. Make it executable, and protect it accordingly.
- When installing PHP as an Apache module:²⁰²
 - Use Apache's authorization features, or design your own in Lightweight Directory Access Protocol (LDAP) (as a very distant, second choice).
 - Do not allow your Apache instance root permissions.
 - Instead of root, use `open_basedir` to control access to directories (and avoid path traversal attacks).
- The PHP session module cannot be counted on alone to secure your session. You must also enable the `session.use_only_cookies`, HTTP Strict Transport Security (HSTS) security header, and `session.use_strict_mode`; use timestamp-based session management; and regenerate session IDs using `session_regenerate_id()` on the regular.²⁰²
- If using user-defined save handlers, ensure that you validate each session ID.²⁰²
- Use `session_destroy()` to destroy old sessions, as they do not automatically self-destruct.²⁰² Ensure that you give it a few seconds before you destroy a session after it becomes inactive by setting a short-term expiration time (timestamp) in `$_SESSION`, and then prohibit access to the session data during that time.
- Use read-only sessions when session data does not need to be updated (this means you don't need to lock it as well, which stops DDOS attacks against the locking).²⁰²
- The current session module does not detect changes to `$_SESSION` when the session is inactive. Don't modify `$_SESSION` when the session is inactive.²⁰²
- Use `output_add_rewrite_var()` for CSRF protection.²⁰²
- Harden your INI (standard configuration file) session settings:²⁰²
 - Set `session.cookie_lifetime=0` to delete the cookie when the browser closes instead of keeping it permanently.
 - Set `session.use_only_cookies=On` to *not* use the default session ID set by GET and POST methods.
 - Use this cookie setting to avoid access from JavaScript: `session.cookie_httponly=On`.

- To ensure that your cookies are only sent over HTTPS: `session.cookie_secure=On`.
- Set your cookie to `session.cookie_samesite="Lax"` or `session.cookie_samesite="Strict"` – prefer strict!
- Perform session garbage collection using `session_gc()`, and combine it with `session.gc_probability` and `session.gc_divisor` to ensure that it deletes obsolete session data. You cannot rely on the garbage collection to happen automatically.
- Disable transparent session IDs as the feature has vulnerabilities in it and sometimes leaks the ID anyway: `session.use_trans_sid=Off`.
- Use `session.cache_limiter=nocache` to avoid caching sensitive data.
- Use `session.hash_function="sha512"` to ensure a strong hashing algorithm.
- When setting `session.save_path`, ensure that you make the directory non-world-readable to avoid insecure direct object references (IDOR).

TIP You may find the setting `session.cookie_httponly=On` to be confusing. If so, you're not the only one! This setting blocks JavaScript from accessing the values inside your cookies, which protects that sensitive data from potential XSS attacks.

- Only allow limited permissions to the PHP web user binary.²⁰² If sending variables to call the web user binary, check the values against an allowlist. Also, double-check for null byte values when performing validation and reject any value that includes null bytes.
- Update often. Make sure you're getting your updates directly from PHP.
- Perform some "security by obscurity" by setting `expose_php` to `off` in your `php.ini` file to hide your PHP.²⁰² You can also change your file extensions (more obscurity) either using a `.htaccess` directive or in the Apache configuration file itself. Perhaps change them all to `".py"` instead?
- Use the password functions provided: `password_hash()` to hash the password and `password_verify()` to verify that the hashed value matches what's stored.²⁰²
- Use either `htmlspecialchars()` or `htmlentities()` to escape special characters that you must accept (if not sanitizing them out). `htmlentities()` escapes all characters, whereas `htmlspecialchars()` just does "special" characters.²⁰³
- Use `filter_var()` to filter user input for validating email addresses, URLs, or IP addresses.²⁰³

- Use `strip_tags()` to remove HTML tags from user input, leaving you with plain text.²⁰³
- Consider trying free SASTs for PHP,²⁰⁴ such as PHPStan, OWASP ASST (Automated Software Security Toolkit), PHP_CodeSniffer, Semgrep Community, progpilot, Psalm, VisualCodeGrepper, and horusec.
- Consider using the Snuffleupagus module, as it kills entire bug classes and provides virtual patching (which really helps with zero days that are being exploited in the wild, such as log4j).²⁰⁵
- When using either the `php.ini`-recommended (first choice) or `php.ini` (second choice) file in production, ensure that you²⁰⁶
 - Change `display_errors` to `off` (it's on by default).
 - Change `display_startup_errors` to `off` (note that this means you need to check the logs).
 - Find a new default temp path. To do so, update this setting: `session.save_path = /tmp/something else`.
 - Set `log_errors = On` to ensure that you are logging.
 - Set `allow_url_fopen` to `false` (it's open by default).
 - Set `ignore_repeated_errors` to `off` (in case you are being attacked).
- Consider compiling your PHP application for your production environment, which means someone with access to your server would not be able to read or modify your application source code.²⁰⁶
- The following are general PHP hardening settings from OWASP; use them.²⁰⁷
 - `doc_root = /path/DocumentRoot/PHP-scripts/`
 - `open_basedir = /path/DocumentRoot/PHP-scripts/`
 - `include_path = /path/PHP-pear/`
 - `extension_dir = /path/PHP-extensions/`
 - `mime_magic.magicfile = /path/PHP-magic.mime`
 - `allow_url_fopen = Off`
 - `allow_url_include = Off`
 - `variables_order = "GPCS"`
 - `allow_webdav_methods = Off`
 - `session.gc_maxlifetime = 600`
- If performing file uploads, use the following settings. If not performing file uploads, turn `file_uploads` off.²⁰⁷
 - `file_uploads = On`

- `upload_tmp_dir = /path/PHP-uploads/`
- `upload_max_filesize = 2M` (or something else reasonable)
- `max_file_uploads = 2` (or something else reasonable)
- Consider disabling dangerous executable functions so you don't need to worry about them:²⁰⁷
 - `enable_dl = Off`
 - `disable_functions = system, exec, shell_exec, passthru, phpinfo, show_source, highlight_file, popen, proc_open, fopen_with_path, dbmopen, dbase_open, putenv, move_uploaded_file, chdir, mkdir, rmdir, chmod, rename, filepro, filepro_rowcount, filepro_retrieve, posix_mkfifo`
- There are several things you can do to lock down PHP further. These came from an OWASP Cheat sheet:²⁰⁷
 - `session.referer_check = /application/path`
 - `memory_limit = 50M`
 - `post_max_size = 20M`
 - `max_execution_time = 60`
 - `report_memleaks = On`
 - `track_errors = Off`
 - `html_errors = Off`
- Use strict equality (`===` and `!==`) whenever you can, rather than `==` (non-strict/regular equality).

What not to do in PHP:

- Do not place any interpreters into `cgi-bin`, such as the PHP interpreter. If you're going to insist on doing that, invoke it as a CGI binary to add extra protections.²⁰²
- Do not count on `session.gc_maxlifetime` to end your session ID. Instead, use timestamp-based expiration.²⁰²
- Do not call `session_regenerate_id(true);` and `session_destroy()` in an active session. It's contradictory and will break the user's session.²⁰²
- Never employ confidential data as a user prefix. Often developers will add the user prefix plus the session ID to make a very helpful GUID. If it's confidential, and you're using it all over the place, it's a disaster waiting to happen.²⁰²
- Developers should not rewrite unneeded HTML tags. It just complicates things, so don't do it unless you really need to.²⁰²

- Use `session.trans_sid_hosts=[list of hosts]` to state which servers can perform `trans_sid` rewrite. Do not put `*` as your list.²⁰²
- To validate a session manually and protect against session ID injection attacks, send the referring page and then validate it when receiving a request: `session.referer_check=[originating URL]`.²⁰²
- Do not send user-supplied input to any of the following functions that execute their arguments: `fwrite()`, `system()`, `exec()`.²⁰²
- Do not use `show_source()`, `highlight_string()`, or `highlight_file()` on a live site; they are only for debugging. They contain sensitive info that could help an attacker.²⁰²
- Do not enable the default error reporting on code running in prod. Instead, use PHP's custom error-handling functions to create your own error handler(s). The default ones leak sensitive system data.²⁰² Consider using the PHP function `error_reporting()` and then evaluating the values of `E_ALL` *in development* to ensure that there's no sensitive data and for testing/debugging. Turn it off before releasing to prod.
- Avoid sending user input to these vulnerable functions: `fopen()`, `fsoc`, `kopen()`, `popen()`, `include()`, `include_once()`, `require()`, `require_once()`, `file_get_contents()`, `imagecreatefromXXX()`, `mkdir()`, `unlink()`, and `rmdir()`.²⁰⁶

As you can see, there's a lot of work to make PHP secure, as it doesn't come with as many tools out of the box as some other languages and frameworks. That said, if PHP is your language of choice, security is a part of that choice. Good luck!

C/C++

The C programming language was first developed in 1972 by Dennis Ritchie.²⁰⁸ Almost as old as the author of this book, C++ was developed starting in 1979 in Bell Laboratories by Bjarne Stroustrup but wasn't available publicly until 1985.²⁰⁹ C++ was created to add OOP to the C language.²⁰⁹ Both are very low-level programming languages, often used for systems that require very fast processing and manual resource management and need to interact with the operating system the software is running on. Neither language is memory-safe, which means that overflow vulnerabilities of all kinds (string, integer, buffer, etc.) abound. You also need to worry about dynamic memory management, manual garbage collection, string formatting complexities, safely reading files, return-oriented programming (ROP), race conditions, pointers, and a lot more. I learned to program in C when I was 16 years old, and because I was in high school at the

time, making apps just for fun, security wasn't an issue for me. Quite frankly, I loved programming right from the start. It made me feel incredibly powerful to make something out of nothing, and I was a natural.

I tip my hat to all those who care for our aging C/C++ applications, helping to make them more secure; much of the backend software across the planet still runs on those languages. Thank you for making our world a safer place.

TIP If you are writing a new piece of software from scratch that requires similar types of processing and flexibility that C and C++ offer, I suggest you strongly consider Rust instead, which is memory-safe. It will save you many headaches.

What to do in C/C++:

- Get a copy of *Secure Coding in C and C++* by Robert Seacord, and then read it.
- Read the *SEI Cert C++*, and consider adopting it for your org as a standard. It's easier to consume than Robert's book (in my humble opinion) but still very thorough.
- If possible, get a static analysis tool to help you secure your code. Be very careful about how well the vendor covers this language in particular, as most of them just have cursory checks. This specific language is very, very difficult compared to other languages for such tools. As of this writing, first-generation static analysis tools perform much better for C/C++ than second gen.
- Decide who will allocate and who will free memory (the caller or the callee), and set that as a standard for each project.²¹⁰ Do not deviate, or you will end up with memory leaks. Ensure that you always free memory when you are done using it.
- Do not use `void*` or type casting to bypass strong type checking. Strong type checking is our friend.²¹¹
- Validate that values are within the ability of that type of variable to hold. This means ensuring that an integer variable value is not larger or smaller than an integer is able to hold (+2147483647). Same with float (1.2E-38 to 3.4E+38) or any other type of variable. C and C++ do not have any safety nets for you, and they will fail mercilessly if you make a miscalculation.²¹¹
- Always initialize your variables. Do not leave it to chance.
- Avoid "off by one" errors. These happen a lot in C/C++. I can attest to this personally.
- Exercise extreme caution when using the following functions. They do not have any safety built into them, and you must double-check your work: `strcpy()`, `strcat()`, `scanf()`, and `gets()`.²¹²

- When compiling C or C++ code on compilers such as GCC and clang, use the following options to have them look for vulnerabilities and errors, and turn on run-time protections:²¹³
 - `-O2 -Wall -Wformat -Wformat=2 -Wconversion -Wimplicit-fallthrough \`
 - `-Werror=format-security \`
 - `-U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=3 \`
 - `-D_GLIBCXX_ASSERTIONS \`
 - `-fstrict-flex-arrays=3 \`
 - `-fstack-clash-protection -fstack-protector-strong \`
 - `-Wl,-z,nodlopen -Wl,-z,noexecstack \`
 - `-Wl,-z,relro -Wl,-z,now`
- Always validate the bounds of any input you receive from the user. That's on top of checking that the type is properly initialized (you may need to add your own null byte at the end) and that it is properly destroyed (memory released and garbage collection) when you're done with it.²¹²
- Use RAII (Resource Acquisition Is Initialization) to automatically acquire resources when an object is created and automatically release them when the object is destroyed. Avoiding potential leaks and errors.²¹⁴
- Use smarter pointers that manage memory allocation and deallocation.²¹⁴
- Ensure that when you call a function, you perform proper error handling. This includes parameter validation.²¹⁴
- Use the *secure* standard libraries, functions, and containers such as `std::vector`, `std::string`, and `std::unordered_map` rather than the older *standard* libraries.²¹²
- When performing expressions (a combination of literals, variables, operators, and function calls that produce a value when evaluated), be careful that they are well-formed and maintained and that you never allow them to return a null or other unintended value. This is an area that needs special attention, according to the CERT C++.²¹²
- When using containers (vectors and arrays), ensure that you do not fall into common traps such as "off by 1" issues, not catching your errors, not validating your bounds, etc.²¹²
- Never read uninitialized memory.²¹²
- Ensure that active locks are released when an exception is thrown.²¹²

What not to do in C/C++:

- Do not pass user-supplied input to executable functions such as `exec()`, `execp()`, and `CreateProcess()`.²¹² If you do use them, be incredibly cautious.
- Do not use `rand()`: it's not random enough.²¹² Instead use `srand()` and feed it the current time, use `random()` (available in C++ 1 and above), or use something less conventional such as `ntheory`, `Math::Prime::Util::GMP`, `Crypt::PRNG`, `Math::Random::Secure`, or `Math::Random::ISAAC`.
- Don't use `printf()`; instead, use the safer `snprintf()` or `fprintf_s()`.²¹²
- Do not leave the file stream open. Close files you have read after you are done.²¹²
- Do not call `setjmp()` or `longjmp()`; use `Throw` and `Catch` instead.²¹²

There are multiple textbooks dedicated to the topic of writing secure C and C++, and this section cannot possibly cover it all. The content covered in this section is just the tip of the iceberg. I suggest that you take some training on this topic and purchase one or more books before you consider yourself ready to write secure C code. I also suggest that you use a static analysis tool and then get a penetration test done on your code before you allow it out into the world.

Conclusion

As we end this chapter on secure coding tips for the 10 most popular programming languages, I hope it is evident that the responsibility of writing secure code lies with each and every software developer. The best and worst practices discussed for languages such as Python, JavaScript, Java, and C++ will equip you to identify and avoid common security vulnerabilities in your code. I am counting on you, the reader, to take the initiative and write better, more secure code. Whether it's being mindful of which functions you use, adopting parts of this chapter as secure coding standards for the place you work, or leveraging more secure libraries, the onus is on every one of us to prioritize secure coding practices.

I encourage every reader to consider further training and/or education in this area. I realize I sound biased because I am a secure coding trainer, but I do this work because it is *needed*. After companies have training, their entire culture changes for the better. The discussions you have as a team, during and after, lead to more secure outcomes for everyone.

Finally, I would like to ask, as a personal favor, that you share at least one thing you learned from this chapter with a friend or colleague. When someone learns from a peer, rather than "Tanya from the security team," they hear it with

different ears, and they take it more seriously. Be that colleague who pushes others to do better.

Chapter Exercises

1. Share one thing you learned from this chapter with a friend or colleague. Do your best to explain to them why it matters and what risk(s) they are avoiding by doing what you've shown them.
2. If you use one of these programming languages as work, when you have some time, pick an application you are responsible for and review it for some of these issues. See if you can find a bug and fix it!
3. If you use one of these programming languages as work, and you do not already have a secure coding guideline or standard, adopt parts of this chapter into a guide for your organization. Yes, I am telling you to copy some of my work to help you and your colleagues write better code. It's okay, I promise.

Popular Frameworks

Programming frameworks provide functionality that helps us program faster and build better apps by providing basic functionalities. Think of a framework as a layer of abstraction that provides tools you can use while you code. Some of the features that frameworks offer include security features, such as output encoding, which can be extremely helpful, taking the onus off the programmer for having to know how to do everything perfectly every single time. That said, some features in frameworks are out of date, have known vulnerabilities, or don't work *quite* as advertised. In this section of the book, we will cover security controls that you should take advantage of, as well as things you should avoid, for several popular programming frameworks.

TIP For every framework, please try hard to keep it up to date as best you can.

Updates bring security fixes, additional security features, and usually other wonderful features as well. Stay on a supported version, at the very least. No matter the framework, keeping up to date is valuable work!

Web and JavaScript

Web frameworks, as you might have imagined, focus mostly on JavaScript. They help make the web front end of your web applications beautiful, interactive,

and fast. Lots of them also have great features that help you avoid cross-site scripting (XSS), which is the number one security issue you want to avoid when it comes to JavaScript.

Express

The Express.js framework (usually called “Express”) is a minimal and flexible framework for Node.js that provides features for web, mobile, and API applications.²¹⁵ Although Express is quite lightweight, it can be augmented via its middleware for even more modules.²¹⁵

TIP Do not put user input into URLs: `href="userProvidedUrl"` . This goes for every JavaScript framework.

TIP You can view the official Express Security Guide here: <https://expressjs.com/en/advanced/best-practice-security.html>.

Security controls to use:

- Use the Helmet module to implement security headers and to help avoid XSS: `$ npm install --save helmet`.²¹⁶
- Don't let people know exactly which version of Express you are using (to avoid fingerprinting) by removing the “powered by” message: `app.disable('x-powered-by')`.²¹⁶
- Add your own “not found” handler and write your own error handler to ensure that you don't share sensitive application error information.²¹⁶
- For cookies, use the `express-session` module over the `cookie-session` module, as the cookie data is visible to the client (not secure). You can also use it to set all your cookie settings, like the `secure` and `httponly` flags.²¹⁶
- Use can use the `rate-limiter-flexible` package to help prevent brute-force attacks. It will help you set limits.²¹⁶
- Use node package manager (NPM) audit to analyze your dependency trees: `$ npm audit`.¹⁸⁷
- Use Snyk CLI for free to see if you have dependencies with known vulnerabilities:¹⁸⁷

```
$ npm install -g snyk
$ snyk test
```

- Sign up for security advisories, such as the one from GitHub at <https://github.com/advisories>.¹⁸⁷
- Avoid regular expressions that are vulnerable to denial of service (DOS) attacks by checking them with <https://github.com/davisjam/vuln-regex-detector>.²¹⁷
- Consider using Hydra (<https://github.com/vanhauser-thc/thc-hydra>) to test brute-force attempts on your login pages. Please only use it to test your own apps; do not use it for evil.²¹⁸ On top of this, consider using `express-rate-limit` or `rate-limiter-flexible` to further protect against brute-force attempts.
- Use the `express-validator` module to validate and sanitize your inputs.²¹⁹
- If using JSON web tokens (JWTs), use `Express-jwt-permissions` with `express-jwt` to ensure that you're not giving away access you should not.²²⁰
- Consider using `sanitize-html` to sanitize your input after validating it.²²⁰

What to avoid in Express:

- Don't use deprecated or vulnerable versions of Express, such as versions 2.x or 3.x; as of this writing, we're on version 4.x and above.²¹⁶
- Do not use the `cookie-session` module for handling your session or anything sensitive.²¹⁶
- Don't use the default cookie name; you can reset it using the `express-session` module.²¹⁶
- Don't use the `ratelimiter` package; instead, use the `rate-limiter-flexible` package.²¹⁶
- Do not trust the built-in output encoding to be "enough" to stop XSS. You must always test with both static and dynamic analysis to ensure that you are safe.

GRC TO THE RESCUE

Last year, Bob worked at a company that was acquired by a much larger company, and he was quite nervous about it. Surprisingly, the acquiring company's GRC (Governance, Risk, and Compliance) team came to the rescue! They helped split off the different products to various teams so they could meet their new colleagues and provided up-to-date documentation on all new processes they needed to follow, including contact information for every team and service, so they never had to wait on anyone. It was like they were a team of social butterflies, helping all the new people settle in. Bob had never experienced GRC like this before, and he was officially a fan.⁹¹

React.js

React is a very popular front-end JavaScript framework that offers powerful built-in security features, such as its virtual DOM implementation, unique JavaScript XML (JSX) syntax, and component-based architecture that encapsulates functionality within components to limit their scope of execution.²²¹ In this section, we will cover security controls you should use, plus features you should avoid or use with extreme caution.

Security controls to use:

- React has a function called `dangerouslySetInnerHTML`, and it is named that to alert developers that what they are about to do is *dangerous*. Ideally, you would not use this feature (which takes user input and renders it), but if you must, then first send your user input through `DOMPurify`, validate it, escape it however you normally would, and then send it to this hazardous function.²²¹
- Consider using the Redux JS package to manage your state in a more consistent manner.²²²
- JSX syntax: Use data binding with curly braces `{}`, and React will automatically escape values to protect against XSS attacks. However, it only works when rendering text content and not when rendering HTML attributes. Only use it for text!²²³ If you do need to put it into HTML attributes that will be rendered (which is a bad idea, by the way), write your own custom input validation, then escaping or sanitization.
- Always ensure that you have HTTPS at the front of links that are taken from the user to avoid URL-based script injection.²²³ Ideally, though, you would check links received from the user against an allowlist and then use the item from the allowlist rather than whatever value you received from the user. Also, use the native URL parsing function to validate the URL.²²³
- React will provide automatic content escaping when using server-side rendering functions such as `ReactDOMServer.renderToString()` and `ReactDOMServer.renderToStaticMarkup()`.²²³ Please use them.
- Escape the `<` character to avoid injection attacks. It looks like this: `/<`.²²³
- When sending JSON data along with server-side rendered React pages, don't forget to escape the `<` character! It looks like this: `JSON.stringify(preloadedState).replace(/</g, '\\u003c')`.²²³ Thank you, Ron Perris and Liran Tal, for the code.
- Use the ESLint React security config; it's a free linter.²²³
- Use JSX syntax rather than concatenating strings with JavaScript.²²¹

- The following libraries use best security practices, and you may want to consider using them: Reactjs Router for routing, Reactjs Hook Form for form validation, Reactjs Redux for state management, and Reactjs Helmet for managing the document head.²²⁴
- You may want to pair your use of React with Express to secure the server side of your app (React just does the front end, but Express can do the backend).
- Consider using the library `secure-json-parse`; it provides additional security checks compared to the default JSON parsing process.²²⁴
- A new concept to me is the *security realm* offered in React. It's essentially an allowlist of all authorized users, and after someone logs in, you compare them against this list. It's an additional layer of security you could add.²²⁵
- You can set `disabled` attributes for any HTML element to make it non-mutable (meaning it cannot change). This means it's not possible to focus on or submit that element within a form. You would need to add validation for the element to enable that attribute if you needed it to work for you again, such as it passing a `user-input-validation` function.²²⁵

What to avoid:

- Don't use deprecated or vulnerable versions of React.js.
- Avoid using user-generated properties with the `createElement` API.²²¹
- Don't use `dangerouslySetInnerHTML`. Just don't. That's the easiest solution.
- React allows developers to access DOM elements directly by using "escape hatches." Never, *ever* put user-supplied data (or data that might have been changed by a user) into an escape hatch. Examples of escape hatches include `findDOMNode` and `createRef`.²²¹
- Do not count on the JSX automatic escaping for anything other than text elements. Anything that could be interpreted (such as any HTML element that is not text) needs to be escaped manually (code you wrote, not JSX).²²³
- Do not concatenate (user-supplied) strings to the output of `renderToStaticMarkup()` before sending the strings to the client for hydration (server-side render rather than client-side render).²²³
- Avoid sending user-supplied or otherwise untrusted data to `renderToStaticMarkup()`.²²³
- Programmers often set the initial state of an app using JSON values. `JSON.stringify()` is a function that converts data into a string, including injected malicious JS objects. Do not use `JSON.stringify()` with user-supplied data.²²³

GOLDEN IMAGES FOR ALICE

Alice's new project had all sorts of legislation that applied to it, as well as internal policies for encryption, networking rules, cloud security, and all sorts of other technical mumbo-jumbo she didn't understand. Alice started to get really worked up until she realized that she knew what to do: talk to her friend Faythe from the GRC team. Faythe explained that they had golden images that already had all the frameworks and benchmarks applied to them, ensuring that they would meet all the regulatory and internet policies Alice needed to comply with. She explained that the testing was already done, they would work well with everyone else's systems, and that Alice could trust her apps were built on secure infrastructure. Alice's stress melted away...⁹¹

Angular

Angular is a popular front-end framework developed and maintained by Google. It offers amazing features such as efficient data binding, dependency injection, modular structure, and seamless integration with TypeScript. Angular offers an extensive library of prebuilt components, dedicated testing utilities, and the ability to create expressive and dynamic user interfaces. It also offers several built-in security features, which we will explore in this section of the book.

Thank you to the Angular framework folks for creating a very well-written security guide, which is heavily referenced in this section and located here: <https://v17.angular.io/guide/security>.

NOTE A *nonce* in cryptography is a randomly generated number used to protect private communications and ensure the authenticity of a transaction.

Security controls to use:

- Angular is part of Google's Open Source Software Vulnerability Reward Program. For vulnerabilities in Angular, please submit your report at <https://bughunters.google.com>.²²⁶ Please only report a bug if you are sure that you have found something; these people are busy. Validate your findings before submitting your bugs.
- Don't alter your copy of Angular. Private, customized versions of Angular tend to fall behind the current version and do not include security fixes and other improvements. Instead, be a great community member and share your Angular changes with everyone by making a pull request.²²⁶
- *Angular treats all values as untrusted by default.* When a value is inserted into the DOM from a template binding or interpolation, Angular sanitizes and escapes the untrusted values. If a value was already validated and sanitized outside of the Angular framework (and thus is considered trusted),

you need to communicate this to Angular by marking the value as trusted.²²⁶ Do this with caution: be certain before circumventing this security feature.

- Angular templates are considered trusted by default and should be treated as executable code, not data. Never create templates with user input added to the template syntax, as this is a recipe for injection vulnerabilities.²²⁶
- Always use the default ahead-of-time (AOT) template compiler (as opposed to the just-in-time [JIT] compiler) for production deployments, to avoid template injection.²²⁶
- As Angular is a JavaScript framework, and JavaScript is the only language XSS applies to, you need to be extremely careful about that specific vulnerability. For extra protection against XSS, configure the Content Security Policy security header to enable Trusted Types enforcement, which will help ensure that your data types stay data and are not interpreted as code by accident (avoiding XSS).²²⁶ Note: this goes for all JavaScript frameworks.
- Angular defines four security contexts:²²⁶
 - HTML: Used when interpreting a value as HTML; for example, when binding to `innerHTML`.
 - Style: Used when binding CSS into the `style` property.
 - URL: Used for URL properties, such as `<a href>`.
 - Resource URL: A URL that is loaded and executed as code; for example, in `<script src>`.

Angular sanitizes untrusted values for HTML, styles, and URLs but *not resource URLs*. Sanitizing resource URLs isn't possible because they contain arbitrary code. In development mode, Angular prints a console warning when it has to change a value during sanitization to let you know it has done it.²²⁶

- Unless you enforce Trusted Types, the built-in browser DOM APIs don't automatically protect you from XSS.²²⁶
- If you are going to mess with the DOM directly, use the built-in Angular sanitization function `DomSanitizer.sanitize`.²²⁶
- If, for legitimate business reasons, you must use executable code in your app that you did not write, and you are certain you have validated and sanitized that input properly, you need to mark it as trusted so that Angular won't undo all your hard work. To mark a value as trusted, inject `DomSanitizer` and call one of the following methods:²²⁶
 - `bypassSecurityTrustHtml`
 - `bypassSecurityTrustScript`
 - `bypassSecurityTrustStyle`

- `bypassSecurityTrustUrl`
- `bypassSecurityTrustResourceUrl`

Be careful that you use the correct context when in this situation: HTML, style, URL, or URL resource.

- The Angular framework pairs beautifully with the Content Security Policy (CSP) header (note: so does every other web framework!), and the following is the absolute bare minimum Angular-framework-recommended implementation of CSP:²²⁶

```
default-src 'self'; style-src 'self' 'nonce-randomNonceGoesHere';
script-src 'self' 'nonce-randomNonceGoesHere';
```

- The nonce for CSP can be set in two ways:²²⁶
 - Set the `ngCspNonce` attribute on the root application element as `<app ngCspNonce="randomNonceGoesHere"></app>`. Use this approach if you have access to server-side templating that can add the nonce both to the header and `index.html` when constructing the response.
 - Provide the nonce using the `CSP_NONCE` injection token. Use this approach if you have access to the nonce at runtime and you want to be able to cache `index.html`.
 - Remember, a nonce is a randomly generated number used to ensure authenticity.
- Enforcing the use of Trusted Types with CSP is recommended by the Angular framework folks and the author of this book.²²⁶ Trusted Types is a web platform that allows applications to lock down powerful APIs to only accept non-spoofable, typed values in place of strings to prevent vulnerabilities caused by using these APIs with attacker-controlled inputs.²²⁷ They help stop XSS by ensuring that the types you created cannot change or ever be treated as code. To enforce Trusted Types for your application, you must configure your application's web server to emit HTTP headers with one of the following Angular policies:²²⁶
 - `angular`: This policy is used in security-reviewed code that is internal to Angular and is required for Angular to function when Trusted Types are enforced. Any inline template values or content sanitized by Angular is treated as safe by this policy.
 - `angular#unsafe-bypass`: This policy is used for applications that use any of the methods in Angular's `DomSanitizer` that bypass security, such as `bypassSecurityTrustHtml`. Any application that uses these methods must enable this policy.

- `angular#unsafe-jit`: This policy is used by the Just-In-Time compiler. You must enable this policy if your application interacts directly with the JIT compiler or is running in JIT mode using the platform browser dynamic.
- `angular#bundler`: This policy is used by the Angular CLI bundler when creating lazy chunk files.
- You need to enable Trusted Types in production, all environments below prod, such as dev (using the Angular CLI), and Karma for unit testing. If you only enable it in prod, you won't be able to test it properly, and it might not go very well for you.²²⁶
- Angular has built-in support to help prevent two common HTTP vulnerabilities: cross-site request forgery (CSRF) and cross-site script inclusion (XSSI):²²⁶
 - **CSRF**: Angular sets the `X-XSRF-TOKEN` HTTP header and sends it with every mutable (POST, PUT, everything but GET) request to relative URLs. You need to check it manually on the backend, though.
 - **XSSI** (also known as JSON vulnerability): Angular's `HttpClient` library automatically strips the string `]]}', \n` that servers add to ensure that the JSON returned is not executable. Ensure that your server is adding that before returning JSON values to your JavaScript.
- Ensure that you review all use of `ElementRef`, which is marked as a security risk.²²⁶ It is the only element in all of Angular that the security folks felt the need to mark as a risk in their security guide. Be careful with it.
- Use interpolation `{{ }}` to safely encode potentially dangerous characters and escape untrusted HTML or CSS expressions within a template expression.²²⁸
- If using user input with an HTML component (which ideally you would not do at all, but if you must . . .), bind its generation to `innerHTML`. This ensures that data will be interpreted as HTML in the correct context and will sanitize the input.²²⁸
- Use the latest, stable version of Angular. Do not use versions below 2, as those are known to be quite insecure.²²⁶
- Angular provides something called *route guards* or *routers*, which protect routes from unauthorized access. They can be used to ensure that only authenticated and authorized users can access various parts of your application. You should use route guards for access control, authentication, and data security. There are multiple route guard classes, such as `AuthGuard` and `RoleGuard`.²²⁹

What to avoid:

- Don't use deprecated or vulnerable versions of Angular.
- Avoid directly interacting with the DOM, and instead use Angular templates where possible. If you must, use the built-in Angular sanitization functions (`DomSanitizer.sanitize` method, with the appropriate `SecurityContext`).²²⁶
- Don't create Angular templates on the server side using a templating language; this carries a high risk of template-injection vulnerabilities. If you must use a templating language, choose one that automatically escapes values to prevent XSS vulnerabilities on the server.²²⁶
- Don't forget to do your half of the CSRF protection by verifying that the `X-XSRF-TOKEN` HTTP header is present on all eligible requests (everything but GET).²²⁶
- Never use native DOM APIs to interact with HTML elements on the page. For example, *do not* do any of the following:
 - `node.appendChild()` ;
 - Use document object methods to interact with the page
 - Use jQuery APIs
- Do not use third-party (non-Angular) template engines (such as Node.JS engines like Pug and Handlebars) to add data or create Angular templates. This is the same as using user input to create an Angular template: bad news for security, as malicious code might be injected.²²⁸

THE SBOM ORDER

Bob read that there was an executive order from the United States that said they had to include an SBOM (software bill of materials) with all new products they delivered to the U.S. government if they wanted to keep their The Federal Risk and Authorization Management Program (FEDRAMP) status. Since the acquisition, Bob knew exactly who to go see: the social butterflies (also known as the GRC team). They gave him a list of supply chain requirements, the SBOM requirements, requirements for the new FEDRAMP sections, and the new National Institute of Standards and Technology (NIST) supply chain "family." They distilled all of it into one manageable list so that it would be easier to apply. This was going to take *way* less time, thanks to the butterflies! Bob bought them donuts to say thank you; he knew where his bread was buttered.⁹¹

jQuery

jQuery is a fast and lightweight JavaScript library used by developers world-wide. With an intuitive syntax and vast plugin ecosystem, jQuery allows you to effortlessly manipulate the DOM, handle events, deploy animations, make AJAX requests, and much more. This flexibility, however, is a double-edged

sword and requires developers to be more diligent when using this framework (this library is often used as a framework, so I will call it that going forward). This framework has existed for a long time and is not used as often as it once was for active development; it also contains very few security features and is not “secure by default” like React, Angular, and Vue.js.²³⁴ In this section, we will cover how to use this older, powerful framework securely.

Security controls to use:

- Always use the latest version of jQuery. Upgrade off versions with known vulnerabilities.
- Use the `text()` method instead of `html()` when inserting text content. This method ensures that the input is treated as plain text, not HTML or script code, blocking opportunities for XSS.²³⁶ This is the only security “feature” offered in this JavaScript framework to combat XSS.
- Validate all responses received in AJAX, and then sanitize and output encode. Don’t put data from an AJAX request into the DOM; treat it as untrusted input.²³⁶
- Carefully validate all input, then sanitize, and then output encode. Write your own routines, as they do not exist in this framework.
- Carefully vet third-party jQuery plugins before using them. A software composition analysis (SCA) tool can help with this.

What to avoid:

- Don’t use deprecated or vulnerable versions of jQuery.
- Although jQuery makes it simple to turn HTML code into elements, you should not do this; instead, you should set the attributes of the HTML. Manually manipulating the HTML elements or the Dom directly is always a bad idea.²³⁵
- jQuery does not include functions for escaping or output encoding; you must write your own manually.²³⁵ You could also use a templating engine like Handlebars that would do the encoding for you automatically.²³⁵

After much research on this topic, avoiding jQuery itself is the best advice the author can offer at this time. If you are using it for backward compatibility, ensure that you update it to the latest version, perform thorough testing (static application security testing [SAST], dynamic application security testing [DAST], SCA, linting), and follow the small amount of advice given in this chapter. Do not use it for new development.

BREAKING UP WITH 90-DAY PASSWORD ROTATION

Alice used to work at a smaller-sized company that forced their employees to change all their passwords every 90 days. It was *so annoying*. Then the security team would walk around and find Post-it notes with passwords written on them at people’s desks

and leave them notes that were . . . not very fun to receive. One day, the company hired its first GRC person. That person immediately told the company to stop. Apparently, NIST said that service accounts (accounts used by computers) needed to be rotated every 90 days, but user account passwords should only change if there's a suspected breach, if the password policy has been updated, or if the password is very weak. She said, "Y'all never needed to do that!" She removed a huge burden from Alice's team and also Alice's memory!

Vue.js

Vue.js is known for its progressive and approachable nature, which facilitates the creation of dynamic user interfaces and single-page applications (SPAs). It also focuses on performance and simplicity and offers a component-based architecture (unlike React and Angular), extremely detailed documentation (again, unlike many other front-end frameworks), reactive two-way data binding, declarative rendering, and more.²³⁰ It's no wonder it's popular. That said, it's a double-edged sword, as SPA applications are very hard (borderline impossible) for traditional dynamic scanners to evaluate; this eliminates one of the three angles of testing we normally do (static, dynamic, and dependencies), which means you need to be even more careful with your coding than usual. Let's dive into Vue.js in this section.

Security controls to use:

- If you find a vulnerability in the Vue.js framework, please report it (security@vuejs.org)!²³¹
- The most fundamental security rule when using Vue is *never to use non-trusted content as your component template*. This is direct from the Vue.js folks, so please pay close attention. Adding untrusted input into a component template is allowing arbitrary JavaScript to execute in your application, which, from an application security perspective, is the absolute worst thing you could do in JavaScript. Even worse, if the code is executed during server-side rendering, it could lead to someone taking over your web server, a data breach, and all sorts of other terrible outcomes.²³¹
- HTML content is automatically escaped by using double curly brackets: `{{ your-content-here }}`.²³¹ This is also known as *v-text*.
- Dynamic attribute bindings are also automatically escaped via native browser APIs (such as `setAttribute`).²³¹
- Services like CodePen and JSFiddle allow user-provided content to be executed, but it's in a context where this is expected and sandboxed (to

some extent) inside iFrames. Exercise caution if sending user input to such services, and test thoroughly.²³¹

- There are no built-in protections for CSRF/XSRF and XSS in Vue.js, so assume you must handle those yourself.²³¹ You can do this by passing a token back and forth with each request and response and then validating that the token is part of the request during each transaction your application makes.
- It is possible to render the view as a client-side SPA *or* render the same HTML components on the server (known as *server-side rendering* [SSR]), then send them to the browser, and then “hydrate” the static markup into a fully interactive app on the client (browser). If you do it this way, a traditional dynamic scanner can take a look at it (unlike the SPA option, which doesn’t really work for most dynamic scanners), which is an advantage from a security perspective.²³² That said, any good penetration tester will be able to test a SPA very thoroughly.
- Use `v-if` (removes an element from the DOM if false) rather than `v-show` (hides the element from the screen and leaves it in the DOM: a less secure option).²³³
- Use `sanitize-url`, `vue-sanitize`, or `vue-3-sanitize` to sanitize URLs rather than doing it manually.²³³
- Use the `:key` directive inside the `v-for` directive to track and identify changes to elements during rendering. This helps to prevent property update attacks (manipulation of the state of a Vue.js component by changing element keys within a `v-for` loop), DOM manipulation, and some types of denial of service (DoS) attacks (the ones that exploit excessive rendering resources).²³³
- Consider using the `zxcvbn.js` library for password complexity rather than attempting to write your own algorithm.

What to avoid:

- Don’t use deprecated or vulnerable versions of Vue.js. Do I sound like a broken record yet?
- It’s best to avoid allowing user input to be used while explicitly rendering HTML content via the `v-html` directive. But if you *absolutely must*, validate/sanitize/escape/encode, test it thoroughly, document why you made this decision, and get sign-off from the security team. This can be done using a template, a render function, or a render function with JSX.²³¹ That said, avoiding this altogether is usually a better security decision.
- Do not put user input into URLs: `href="userProvidedUrl"`. This goes for every JavaScript framework, not just Vue.js.²³¹

- Do not put user input into styles—e.g., `style="userProvidedStyles"`.²³¹ If you must, use `{ { } }` to escape the value, validate and sanitize the URL you're getting it from, and only do this within a sandboxed `iFrame`.²³¹
- Don't render `<script>` tags in Vue. Just don't. Your life will be better.²³¹
- Never mount Vue on nodes that may contain server-rendered and user-provided content.²³¹ It's impossible for Vue to know what the server-rendered content is, and therefore, it cannot protect you from it.

COMPLICATED CERTIFICATES

Bob used to manage a team that managed certificates for their own apps. Every team had to manage their own certificates, and sometimes mistakes were made. They had to rotate *every* year because NIST said so. One day, the GRC team told Bob they could rotate every other year because NIST allowed for that, and they offered to fill out the extension requests for his team as well to save time. The GRC team realized that they could reduce complexity by helping all the teams that needed to manage certificates in a centralized way, and that saved money and time. Plus, the auditors liked it. Bob was very pleased that he didn't have to worry about this as much anymore.⁹¹

Other Frameworks and Libraries

There are many types of frameworks, and code is not only written for the Web in JavaScript! In this section, we will mostly cover frameworks that do not focus only on the Web and/or JavaScript.

.NET (Core)

When I worked for the Canadian Federal Public Service, I wrote *a lot* of .NET, mostly in the VB programming language but also some C#. The first time I had a vulnerability scan run on my code, now known as DAST (an automated tool for finding bugs in web apps), my app was written in .NET, and the scan found XSS. At the time (2011), I had never heard of XSS; when I searched the Web, I remember finding only three websites that mentioned it, and one was OWASP. It took me two more tries to "pass" my scan, and I recall that we even had a team-wide meeting to discuss XSS and what we were going to do about it. I worked with some really smart people at the time, and we built a dynamically linked library (DLL, a library for .NET) that would do input validation and output encoding for us from then on. Back then, .NET did not have as many amazing security features as it does now. We also made a template for all new apps, which included the DLL as well as a well-tested login page and other

security functionality. We shared lessons learned and had great documentation, and honestly it was one of the best development teams I ever worked on. But not every team is like that, and not every organization is that well organized. In this section, we will cover how you and your team can take advantage of this powerful, security-oriented, versatile framework.

Note: please read the C# section in Chapter 6, as there is quite a lot of .NET framework mixed in with the C# advice.

Security controls to use:

- Receive security notifications by selecting the Watch button in GitHub at the following repositories:²⁴⁰
 - .NET Core Security Announcements
 - ASP.NET Core & Entity Framework Core Security Announcements
- Whenever possible, use the built-in security features of .NET, such as the authentication and session management features, rather than writing your own.
- Check on every single page and resource (such as an object or piece of data) if someone should have access or not, using `System.Web.Security.Roles.IsUserInRole(userName, roleName)` or whatever other function you have for that. Ideally, you would use the built-in ones.²⁴⁰
- Use `SecurityEssentials/Core/HttpHeaders.cs` to implement security headers if using .NET Core (which has been deprecated).²⁴⁰
- Consider using `dionach/StripHeaders` from GitHub to strip out security header information in the responses. You can also disable the information via `web.config` or `Startup.cs`, again, to give away less information.²⁴⁰
- Use the Windows Data Protection API (DPAPI) for secure local storage of sensitive data.²⁴⁰
- Use the Entity Framework as your ORM (object relationship mapping), a framework for your database. Also, use stored procedures (MS SQL's parameterized queries) to interact with the database, not inline or dynamic SQL.²⁴⁰
- .NET 5 + supports `ProcessStartInfo.ArgumentList`, which performs *some* character escaping, but it is far from perfect. Use it, but also validate and sanitize.²⁴⁰
- Ensure that debug and trace are off in production. This can be enforced using `web.config` transforms:²⁴⁰
 - `<compilation xdt:Transform="RemoveAttributes(debug)" />`
 - `<trace enabled="false" xdt:Transform="Replace"/>`

- .NET provides anti-forgery tokens to prevent CSRF; ensure that you use the `IAntiforgeryAdditionalDataProvider` type in .NET Core. There are also offerings for AJAX, Windows, etc.²⁴⁰
- Use ASP.NET (Core) Identity features to manage identity.²⁴⁰
- Ensure that you digitally sign assemblies and executable files and use NuGet package signing.²⁴⁰
- Sign and/or encrypt serialized objects to ensure their integrity. Validate the signatures and/or perform an integrity check on encrypted serialized objects before deserializing. Do not accept serialized objects from unknown sources for any reason.²⁴⁰
- For data processing, use one of the following safer processors:²⁴⁰
 - `XmlSerializer` and `DataContractSerializer` to serialize object graphs into and from XML
 - `BinaryReader` and `BinaryWriter` for XML and JSON
 - The `System.Text.Json` APIs to serialize object graphs into JSON
- .NET provides a logging function that you may find handy because, of course, you are logging the outcome of every call to each security control: `Microsoft.Extensions.Logging`.²⁴⁰
- Considering using Application Insights for monitoring.²⁴⁰
- Use `IPAddress.TryParse()` and `Uri.CheckHostName()` to ensure that IP addresses and domain names are valid.²⁴⁰
- Use the Microsoft AntiXSS library when using older (4.5–4.9x) versions of .NET: `Install-Package AntiXSS`.²⁴⁰ It is not available for newer versions 5 and above.
- Use custom error pages, and avoid showing the user any system data: `<customErrors mode="On" defaultRedirect="ErrorPage.html"/>`.²⁴⁰
- Applications can use the `IAuthentication` service to access third-party authentication services such as X (Twitter), Google, and in-house OAuth services.²⁴¹
- .NET compilers compile code into Microsoft Intermediate Language (MSIL), which enforces *managed execution*, meaning it can only access authorized memory locations. This ensures proper isolation between modules and verifies that the runtime can enforce security restrictions. Microsoft calls this “type safety” despite it having nothing to do with what the rest of the industry refers to as *type safety*.²⁴¹ Please do not turn this off or otherwise circumvent it.
- If you must handle untrusted code, use one or more of the following security measures: virtualization, AppContainers, Hyper-V containers,

and/or restrictive operating system (OS) users and permissions.²³⁷ Always validate and then escape or sanitize untrusted code, ideally against an allowlist.

- Protect your data while it's in memory by declaring it as private or internal (and limit scope to the same assembly).²³⁷
- Use .NET's built-in role-based security to implement authorization within your apps.²³⁸
- .NET supports three kinds of principals in the role-based security features it offers: general (nothing to do with Windows), Windows (everything to do with Windows users and roles), and custom (defined however you like).²³⁸ Use the `System.Security.Permissions.PrincipalPermission` class to choose which one you want, and then use that feature to validate permissions (access control/authorization) throughout your app.²³⁸
- For cryptographic functions:²³⁹
 - Use the `RandomNumberGenerator` class.
 - Use .NET *one-shot* APIs for hashing and hash-based message authentication code (HMAC), such as `SHA256.HashData`, unless you need to add customization or have a business requirement to write your own implementations. Only write your own implementations if you absolutely have to.
 - If you need to write your own or implement your own encryption algorithms, ensure that you inherit from one of the algorithm type classes: `SymmetricAlgorithm`, `AsymmetricAlgorithm`, or `HashAlgorithm`.
 - In most cases, you don't need to directly reference an algorithm implementation class, such as `AesCryptoServiceProvider`. The methods and properties you typically need are on the base algorithm class, such as AES. Whenever possible, do not invent your own.

Microsoft-recommended algorithms class by application:²³⁹

- Data privacy : AES
- Data integrity: HMAC-SHA256, HMAC-SHA512
- Digital signature : ECDSA, RSA
- Key exchange: ECDH (Elliptic-Curve Diffie-Hellman), RSA
- Random number generation: `RandomNumberGenerator.GetBytes`, `RandomNumberGenerator.Fill`
- Generating a key from a password: `Rfc2898DeriveBytes.Pbkdf2`
- Encrypt sensitive parts of the `web.config` using `aspnet_regiis -pe`.²⁴⁰

- Remove the version header by adding this to `Machine.config`: `<httpRuntime enableVersionHeader="false" />`.²⁴⁰
- Remove the Server header using the `HttpContext` class in your code: `HttpContext.Current.Response.Headers.Remove("Server");`.²⁴⁰
- Validate URIs using `Uri.IsWellFormedUriString`.²⁴⁰
- Compare each user-provided username with `User.Identity.Name`. They must match.²⁴⁰
- Check roles against `User.Identity.IsInRole` before giving access to anything.²⁴⁰
- When using model binding, ensure that either (1) you intend to assign 100% of the values from your form or (2) you assign only the ones you mean to, to avoid overbinding/mass assignment model-binding attacks (where you intend to update only some of the values in an object, but the attacker adds extra values to the input, and then your app unknowingly assigns the attackers values to the extra fields you meant to leave out).²⁴² You can do this by explicitly listing each property you want to update, marking a property you don't want changed as read-only for when first initializing values, or the `TryUpdateModel` model for when updating values.

Although there are certainly more security features in this robust framework, hopefully, this list gets you started on the right path.

What to avoid:

- Do not use Code Access Security (CAS).²³⁷
- Do not use partial trusted code.²³⁷
- Do not use the `AllowPartiallyTrustedCaller` attribute (APTCA).²³⁷
- Do not use .NET Remoting.²³⁷
- Do not use Distributed Component Object Model (DCOM).²³⁷
- Do not use binary formatters.²³⁷
- Do not use the `BinaryFormatter` type, which is dangerous and not recommended.²⁴⁰
- Avoid or proceed with extreme caution when using the `[AllowHTML]` attribute or helper class `@Html.Raw`.²⁴⁰ Do not use them with untrusted data.
- Do not use `SqlCommand` with user data or untrusted data.²⁴⁰
- If using Internet information services (ISS), do not use tracing (turn it off) in production.²⁴⁰
- Don't use `Eval()`. Generally.²⁴¹
- Avoid overposting/mass assignment model-binding attacks²⁴² by using the `BIND` attribute when first initializing values or the `TryUpdateModel` model when updating values.

MULTIAPPROVAL BLUES

Alice needed what looked like 10 approvals for her new app to go to prod. She was exhausted just *looking* at the list. She decided to try her luck and went to see GRC. She asked them if all those people really needed to approve the release or if perhaps some of them just needed to be *informed* of the release. Faythe came through for her again. Faythe told her this extensive approval process negatively affected several teams and seemed like perhaps it was wasting a lot of company time and resources and *not* adding value. A month later, Faythe met with Alice and presented the new process: a list of five people who needed to approve brand-new changes (new apps or rewrites) and five who needed to be notified. If the change was smaller (bug fixes or only one new feature), and it passed all the checks in continuous integration and continuous delivery/deployment (CI/CD) (SAST, SCA, and linting), then she needed one manual approval and three automated approvals (passing tests) and to send nine notifications. Faythe saved her time again! Alice made a secret plan to take Faythe to lunch one day; she needed to get to know this lady better.

Ruby on Rails

Ruby on Rails, often referred to as Rails, is a full-stack framework built on the powerful Ruby language. Rails embraces principles of convention over configuration and don't repeat yourself (DRY) to streamline development, allowing developers to focus on innovative features rather than boilerplate code. With its rich ecosystem of rubygems (libraries), built-in support for testing, and emphasis on security best practices, Rails remains a favorite for startups and enterprises alike seeking rapid development and scalable web applications. In this section, we will dive into the dos and do nots of Rails.

Security controls to use:

- Rails provides a session object for each user on an application. Session IDs are passed in secure cookies and stored in one of these places:²⁴³
 - `ActionDispatch::Session::CookieStore`: Stores everything on the client
 - `ActionDispatch::Session::CacheStore`: Stores the data in the Rails cache
 - `ActionDispatch::Session::MemCacheStore`: Stores the data in a memcached cluster (this is a legacy implementation; consider using `CacheStore` instead)
 - `ActionDispatch::Session::ActiveRecordStore`: Stores the data in a database using ActiveRecord (requires the `activerecord-session_store` rubygem)
 - A custom store or a store provided by a third-party rubygem

- Rails uses `ActionDispatch::Session::CookieStore` as the default session storage. The `CookieStore` uses the encrypted cookie jar to provide a secure, encrypted location to store session data. The `CookieStore` can store around 4 KB of data, which isn't much, but it's usually enough.²⁴³
- Use `bin/rails secret` to get new, long, random, and unique secrets.²⁴³
- For creating a cryptographic salt value, in test and development applications, you can get a `secret_key_base` derived from the app name. Other environments must use a random key present in `config/credentials.yml.enc`.²⁴³
- Don't forget to create a new session ID and invalidate the old one whenever anyone logs out or times out: `reset_session`.²⁴³
- Consider using the rubygem `Devise` or `AuthLogic` to provide authentication. If you want to allow users to log in via third parties such as LinkedIn, Facebook, Google, and so on, you can use the `omniauth` rubygem.²⁴⁴ `Devise` will also handle user management and other security for you.²⁴³ You can also use the built-in `has_secure_password` method, which supports secure password hashing, confirmation, and recovery mechanisms.²⁴³
- An anti-CSRF token is passed automatically as a default. That said, you should probably verify that it has not been disabled by accident:²⁴³
`config.action_controller.default_protect_from_forgery = true.`
- By default, Rails logs all requests being made to the web application, including sensitive ones. Use the `logging` filter to ensure that you don't log anything you shouldn't: `config.filter_parameters << :name-of-your-secret`.²⁴³
- Regular expressions work slightly differently in Ruby. To avoid XSS and other injections, ensure that you use `\A` (start) and `\z` (finish) instead of `^` (start) and `$` (finish).²⁴³
- Apply authorization checks when getting records using the `current_user` function. This ensures that you validate that the user is allowed to see whatever resource you are fetching:

`@project = @current_user.projects.find(params[:id]).`²⁴³
- Allow `<strong>` instead of removing `<script>` to protect against XSS.²⁴³ Ruby on Rails has a built-in filter for special SQL characters, which will escape `'`, `"`, the NULL character, and line breaks. If using `Model.find(id)` or `Model.find_by_something(something)`, it automatically applies this countermeasure. But in SQL fragments, especially in condition fragments

(where("."...")) and the `connection.execute()` and `Model.find_by_sql()` methods, it has to be applied manually. Rail's feature used in place of parameterized queries is called *positional handlers*. `Model.where("zip_code = ? AND quantity >= ?", entered_zip_code, entered_quantity).first.`²⁴³ Note: this only works with model instances.

- Consider using the ORM called `ActiveRecord`; it will help you avoid SQL injection.²⁴⁴
- If not working with model instances (the earlier tip), use `sanitize_sql` to sanitize user input before sending it to a database.²⁴³
- When sanitizing user input for HTML, use the Rails method `sanitize()`.²⁴³
- For output encoding, use the `html_escape()` (or its alias, `h()`) method to replace the HTML input characters `&`, `"`, `<`, and `>` by their uninterpreted representations in HTML (`&`, `"`, `<`, and `>`).²⁴³
- If you wish to obfuscate and encode strings, use `Hackvector`.²⁴³
- If you use the `system` command, use it with parameters to pass the parameters safely: `system(command, parameters)`.²⁴³
- Use the `ActionDispatch::HostAuthorization` middleware to guard against DNS rebinding and other Host header attacks. It is enabled by default in the development environment, but you need to activate it in production and other environments by setting the list of allowed hosts.²⁴³
- The following security headers are configured by default in Rails:²⁴³
 - `'X-Frame-Options' => 'SAMEORIGIN'`,
 - `'X-XSS-Protection' => '0'`,
 - `'X-Content-Type-Options' => 'nosniff'`,
 - `'X-Permitted-Cross-Domain-Policies' => 'none'`,
 - `'Referrer-Policy' => 'strict-origin-when-cross-origin'`
- You can (and should) add even more security headers in `config/application.rb`.²⁴³
- Rails stores secrets in encrypted format in `config/credentials.yml.enc`. Rails uses `config/master.key` or alternatively looks for the environment variable `ENV["RAILS_MASTER_KEY"]` to encrypt the credentials file. The credentials file contains the application's `secret_key_base`. It can be stored in source control, but the master key must be kept elsewhere/safe.²⁴³ Or you can use a secret management tool.
- Secrets kept in the credentials file are accessible via `Rails.application.credentials`.²⁴³

- Use `bundle update --conservative rubygem_name` to safely update vulnerable dependencies.²⁴³
- Use Brakeman, a free static analysis tool built only for Ruby, to find security issues in your code.²⁴⁴ It's not perfect, but it's a fantastic start, especially because it's free!
- By default, cookies persist forever in Ruby. You don't want this for session cookies or anything else sensitive. Ensure that you set a reasonable timeout for them to self-destruct.²⁴⁴
- To enforce password complexity, consider using the `zxcvbn` rubygem.²⁴⁴
- Paths are intuitive and very guessable in Ruby. To protect users from having someone attempt to access their records, you need to implement access control. Consider resource-based access control libraries such as `cancancan` and `pundit` for this purpose.²⁴⁴
- Examine your attack surface (the areas of your code accessible to an attacker) by looking in the `/config/routes.rb` file. It dictates what URLs should be accessible and handled by which controllers. You want to ensure that only the URLs you intend to be accessible to the public/users are accessible, and you can look in this file to cross-reference the controllers and URLs versus what you think your users should be allowed to access.²⁴⁴ On top of this, remove any routes or API actions you don't want to expose. Rails auto-generation is incredibly helpful, but at times, it can create an unnecessary (unused) surface area, which should be removed.
- If using the Active Record ORM framework for databases, exercise caution when using the following methods, as they do not sanitize the input for SQLi: `Calculation`, `Delete All`, `Destroy All`, `Exists?`, `Joins`, `Order`, and some others.²⁴⁵
- To encode strings (such as URLs) manually, use `url_encode(s)` or `u(s)`.²⁴⁵
- Consider using the `secure_headers` rubygem, created by the Twitter security team, to implement additional security headers.²⁴⁵
- Sign your cookies containing your session ID using `cookies.signed[:secure_session]`. Signing adds an additional layer of security to ensure the integrity of the cookie.²⁴⁵
- Force TLS, set the `secure` flag on your cookies, and redirect to HTTPS, all by adding `config.force_ssl = true` to your `config/application.rb` file.²⁴⁵

What to avoid:

- Do not disable the built-in security defaults unless you have an extremely good reason. If you do so, document it and test it thoroughly. Example:

```
config.action_controller.default_protect_from_forgery = false.
```

- Do not use the `html_safe` function willy-nilly. The `html_safe` function means that you have properly validated/sanitized/escaped/encoded the information you are putting into that field. You are instructing Ruby *not* to encode this information. You are *removing* the built-in default security. Only use this feature if you are *absolutely sure* you know what you are doing.²⁴⁴
- Process media files asynchronously to avoid DOS attacks.²⁴³
- If you use RedCloth, you must use it with an allowlist input filter, as it still has some security issues.²⁴³
- If you use the `in_place_editor` plugin or actions that return a string rather than rendering a view, you must escape the return value in the action using the `h()` method.²⁴³
- Exercise extreme caution with the following functions, or better yet, avoid them altogether: `system(command)`, `exec(command)`, `spawn(command)`, and `command`.²⁴³
- Do not use `Kernel#open`. Instead, use `File.open`, `IO.open`, or `URI#open`.²⁴³
- Do not build response headers based on user input. If you must, validate and sanitize or escape the input, and then encode it.²⁴³
- Do not disable `deep_munge`, as it can cause unexpected database behaviors. It is on by default: `config.action_dispatch.perform_deep_munge = true`.²⁴³
- Never submit user input to the following commands. If you absolutely must, validate the input, sanitize or escape it, encode it, and then test it extremely thoroughly. But it's better if you don't do it at all: `Eval()`, `System()`, `ls -al /`, `exec()`, `spawn()`, `open()`, `Process.exec()`, `Process.spawn()`, `IO.binread()`, `IO.binwrite()`, `IO.foreach()`, `IO.popen()`, `IO.read()`, `IO.readlines()`, `IO.write()`.²⁴⁴
- Fields that use `raw`, `html_safe`, `content_tag` or similar are potential XSS targets. Validate/sanitize/escape/encode if you are putting untrusted input into any of these. Avoid them altogether if possible.²⁴⁴
- Do not send transactions on GET requests in any language, as it does not have built-in protections like the POST method. Rails does not provide CSRF protection for any HTTP GET request, for example.²⁴⁴
- Exercise extreme caution with the `render` method, and never send it untrusted data.²⁴⁴
- Avoid checking the following files into source control, as there is a strong likelihood that they contain sensitive data. Or, encrypt them before saving them: `/config/database.yml`, `/config/initializers/secret_token.rb`, `/db/seeds..rb`, `/db/development.sqlite3`.²⁴⁴

- Consider using the `rubygem Devise`, which uses `bcrypt` for password hashing, rather than the default OS encryption.²⁴⁴
- Other free tools you can use to test your Ruby include `Semgrep OSS`, `Bearer`, `Gauntlt`, and the `dawnscanner rubygem`.

COMPLIANCE != SECURITY

Bob used to think compliance was stupid. *Compliance != security*, in his opinion. For compliance reasons, his team spent a lot of time documenting things and creating templates at the new company, and at first, he wasn't sure it was time well spent. Then one day they had a very scary bug submitted via their bug bounty program. Thanks to all the documentation, they figured out in about 5 minutes where this code was used across all their applications. They took the template out, fixed the bug in it, tested it, and then shared the fix across all their apps. It was *amazing*! Bob couldn't believe how efficient and easy everything was. At his old company, this could have been a 6-month project for three people, and his team addressed it in one long day. Although Bob was not exactly the number-one fan of compliance afterward, he certainly had a more positive outlook from then on.

Spring and Spring Boot

Spring, a Java-based framework, has superseded the wildly insecure Java Struts framework. Spring Boot, the leading Java framework, is an extension of the original Spring framework.²⁴⁶ Spring Boot seamlessly integrates with the Spring ecosystem and offers robust features such as autoconfiguration and standalone deployment, which makes for streamlined and efficient building of production-grade, enterprise-ready, scalable applications. It supports microservices architecture, offers built-in security features, and is often used for rapid prototyping. This section will cover both Spring and Spring Boot security features, gotchas, and free tools.

WHY I SPEAK POORLY OF STRUTS

Java Struts made my life extremely difficult throughout 2016 and 2017, thanks to the remote code execution (RCE) vulnerability found in multiple versions of the framework and the fact that I was charged with launching an AppSec program at a place that had hundreds, if not thousands, of Struts apps. It caused numerous stressful days and sleepless nights. Struts also has fewer and weaker security features than Spring and Spring Boot. As a result, I strongly recommend Spring or Spring Boot over Struts.

LOG4J

There are several versions of the Java logging library named Log4j that have very serious (RCE) vulnerabilities that can be exploited just by adding some text at the end of the URL in the address bar of your browser. The attacker doesn't need to be logged in. They don't need to write a lot of complicated code; the attack string is available all over the internet. The attack is unbelievably simple and terrifying. If you are using Log4j, ensure that you are using version 2.18 or above, but ideally something much newer (such as 2.24, which was current as of this writing). Companies all over the world are still being attacked by this vulnerability from 2021. Don't be one of them!

Security controls to use:

- Use the Spring Security framework for authentication and access control.²⁴⁷
- Use JDBC as your ORM framework for safer DB access.
- Use Spring Session to manage sessions.²⁴⁷
- If you are a Hashicorp vault customer, you can use Spring Vault to access it easily and out of the box.²⁴⁷
- Spring comes with CSRF protection by default. If using a JavaScript framework for your front end and Spring for the backend, you will need to configure the `CookieCsrfTokenRepository` so the JavaScript can read the cookie. If you are not using a JavaScript framework, Spring MVC's `<form:form>` tag or Thymeleaf and `@EnableWebSecurity` will automatically add the CSRF token as a hidden input field.²⁴⁸
- Spring security also provides a number of security headers by default: `Cache-Control`, `Pragma: no-cache`, `Expires`, `X-Content-Type-Options`, `Strict-Transport-Security`, `X-Frame-Options`, and `X-XSS-Protection`.²⁴⁸ That said, you should add all applicable headers, as outlined previously in this book.
- Use the `PasswordEncoder` module within Spring Security to hash passwords.²⁴⁸
- The Spring Security framework also integrates with cloud platforms, such as Kubernetes and Cloud Foundry, to provide additional security features, including mutual TLS authentication and service-to-service authentication.²⁴⁹
- Use the Spring Boot `Actuator` module to monitor your microservices and log security events.²⁴⁹
- The `@Secured` annotation provided by Spring Security allows developers to specify which roles are required to access a particular method.²⁴⁹

- The `@PreAuthorize` and `@PostAuthorize` annotations allow developers to specify which roles are required to access a particular method, both before and after the method is executed. `@PreAuthorize` and `@PostAuthorize` can be used together to provide both pre- and post-method security.²⁴⁹
- Spring security offers security filters, which you can customize (add even more strict checks) by implementing `javax.servlet.Filter`.²⁵⁰
- In Spring Security, security providers are pluggable authentication mechanisms responsible for authenticating users, managing user credentials, and establishing an authenticated session. Providers include:²⁵⁰
 - `InMemoryUserDetailsManager`: Stores user credentials and roles in memory (does not scale well)
 - `JdbcUserDetailsManager`: Retrieves user credentials and roles from a database table
 - LDAP authentication provider: Authenticates users against an LDAP server
 - OAuth2 authentication provider: Authenticates users using OAuth2, allowing users to sign in using their social media or Google accounts
 - OpenID Connect authentication provider: Authenticates users with OpenID Connect, an authentication layer on top of OAuth2
 - Anonymous authentication provider: Allows unauthenticated users to access certain resources without the need to log in

What to avoid:

- Do not disable security features such as `.csrf().disable()`; This is a mistake.²⁴⁹
- Avoid using `@PermitAll`. It may expose endpoints or data to unauthorized users.

IMPOSSIBLE POLICIES

Alice was frustrated. She found what she felt was an impossible policy. It was based on FEDRAMP and NIST, assurance level 3. It demanded evidence that her company could prove they held the key for OpenID Connect, but her engineers were telling her they couldn't have the key; that's not the way it's built. She even made them read the request for change (RFC), which says you cannot manage the keys yourself. Alice marched down to the GRC office, planning to blow a gasket on them. She decided to walk around the building first to calm down so she could be patient and professional despite wanting to scream. When she explained the situation to the GRC folks, they said that the requirement she was concerned about was a "forward-looking requirement," meaning that it was a plan for the future, assuming systems would be able to meet it. Alice was confused, "So what do we do? I can't meet this requirement."

It's impossible." They said, "We will change the policy to meet what's possible. Don't worry; we will fix it. Thank you for bringing it to our attention." And that was that. Alice had prepared for battle . . . but it turns out logic was all she needed!⁹¹

Flask

Flask is a simple, flexible, and lightweight Python web microframework that is a wildly popular choice for web development. When we add the Python package `flask-security` to it, Flask becomes even more powerful! With its lightweight and minimalist design, Flask provides a balance between simplicity and functionality, allowing developers to focus on building rather than complexity. Its extensive ecosystem of extensions, combined with vigorous security features, make Flask + Flask-Security an excellent framework for developing secure and scalable web applications. Let's explore its security features together in this section.

Security controls to use:

- Flask configures Jinja2 to automatically escape all values unless explicitly told otherwise. That said, it cannot protect you from XSS by attribute injection. Do not put untrusted data into HTML attributes, and ensure that you double-quote the values you do add, like so: `<input value="{ value }">`.²⁵¹
- Use the anti-CSRF protection: the Flask-WTF extension, with `CSRFProtect` middleware.²⁵¹
- Consider using Flask authentication libraries such as `Flask-Login` and `Flask-JWT`.²⁵¹
- Flask offers its own secret management tool, `Flask-Secrets`.²⁵¹
- As always, you should use parameterized queries to make calls to the database, but in Flask, you can also use the `execute` method of the cursor object to bind parameters securely. You still need to validate and escape or sanitize the values you put in there.
- Consider using some of the following Flask libraries for security: `Flask-Login` (user session management), `Flask-Principal` (identity management), `Flask-WTF` (integration of Flask and WTForms, including CSRF, file upload, and reCAPTCHA), `itsdangerous` (signs data to validate integrity), `passlib` (password hashing), `webauthn` (makes it easy to leverage the power of WebAuthn), and `authlib` (builds OAuth and OpenID Connect servers: JWS, JWK, JWA, JWT included).²⁵²

- Two-factor authentication can be enabled using the user's `totp` secret, a time-based, one-time password.²⁵²
- WebAuthn connects authenticators (such as YubiKey and mobile biometrics) with websites. Flask-Security supports using WebAuthn keys as either first or secondary authenticators.²⁵²
- Flask-Security also offers email confirmation, password reset/recovery, email validation, user registration, password changes, and login tracking, all out of the box.²⁵²
- Flask-Security provides authentication decorators:
 - `auth_required()`
 - `http_auth_required()`
 - `auth_token_required()`
- Flask-Security also provides authorization decorators:
 - `roles_required()`
 - `roles_accepted()`
 - `permissions_required()`
 - `permissions_accepted()`
- The authorization and authentication decorators are simple and unlikely to satisfy all of your requirements; you will probably need to use a special exception class that you can raise either in response to Flask-Security authorization failures or in your own code.²⁵²
- By default, the Flask-Security endpoints that require the caller to be authenticated do *not* support Basic Auth.²⁵²
- Flask-Security supports requiring a “fresh” or recent authentication via `auth_required()` (parameters define how recent the authentication must have happened), `Security.reauthn_handler()` (called when a request fails the recent authentication check), and endpoints that request the user to reauthenticate (`SECURITY_VERIFY_URL`, `SECURITY_US_VERIFY_URL`, `SECURITY_WAN_VERIFY_URL`). Several of the default Flask endpoints (`.wan_register`, `.wan_delete`, `.tf_setup`, `.us_setup`, and `.mf_recovery_codes`) can also be set for “freshness” via the `SECURITY_FRESHNESS` and `SECURITY_FRESHNESS_GRACE_PERIOD` configuration variables.²⁵²
- Flask-Security has a default password validator that²⁵²
 - Checks for minimum and maximum length (minimum is configurable via `SECURITY_PASSWORD_LENGTH_MIN`).
 - If `SECURITY_PASSWORD_CHECK_BREACHED` is set, it uses the API for haveibeenpwned to check if the password is on a list of breached passwords.

The configuration variable `SECURITY_PASSWORD_BREACHED_COUNT` can be used to set the minimum allowable “breaches.” Thank you to Troy Hunt and his team for making this available.

- Password validation and Unicode normalization functionality are offered in `PasswordUtil`.²⁵²
- Flask-Security sets the `SECURITY_RETURN_GENERIC_RESPONSES` configuration to `True` by default, which ensures that generic responses are sent that do not allow for user enumeration.²⁵²
- Set `SECURITY_CONFIRMABLE` to `True` to avoid email enumeration.²⁵²
- Use the `WTForms` library to create form objects with defined fields and validation rules to build your input validation directly into your form.²⁵³
- To sanitize HTML output in your Flask app, you can use the `Bleach` library.²⁵³
- Use `flask.escape()` to escape sensitive characters.²⁵⁴
- The `Flask-SSLify` extension enforces HTTPS.²⁵⁵
- Consider `Flask-JWT-Extended` for token-based authentication.²⁵⁵
- Consider `Flask-Limiter` for rate limiting to protect against form abuse, DDoS attacks, and excessive API usage.²⁵⁵
- If you are allowing file uploads, just as you would for any language or framework, do not use the name provided by your user; select your own. If for some reason you must use the name provided by your user, consider using the `secure_filename()` function provided by the `Werkzeug` library to try to avoid the directory traversal vulnerability.

What to avoid:

- Do not use `@login_required` if you plan to use Flask-Security; it will do that for you/they conflict.²⁵²
- Do not put user input into the `next` parameter, as it could cause open redirects for the following endpoints: `.login`, `.logout`, `.register`, `.verify`, `.two_factor_token_validation`, `.wan_verify_response`, `.wan_signin_response`, `.us_signin`, and `us_verify`.²⁵²
- If your application is using `WebAuthn`, set `SECURITY_WAN_ALLOW_USER_HINTS` to `False` to avoid allowing for username and email enumeration.²⁵²
- When using Jinja2, avoid using the `|safe` filter or `Markup` class unless absolutely necessary, as these can make your application vulnerable to XSS by marking a string as safe HTML.²⁵⁵
- Avoid generating HTML templates without using Jinja2.

NEVER STOP LEARNING

Bob realized early in his career that if you want to be great at any part of the technology field, you need to love learning; there's just no way around it. From the GRC folks, to his amazing programmers, to the AppSec folks and sysadmins, every team seemed to need to learn all the time. He decided to set up quarterly check-ins with his team to set goals for learning and told his boss he wanted to do that, too. Each quarter, they would learn something new, whether technical or soft skills. There was always a way to improve, and Bob wanted to enable his team to be the best they could be.

Chapter Exercises

1. If you use one of these frameworks at work, create a presentation, write up a summary, or create a guideline on that framework and share it with your colleagues. They will thank you!
2. Try to use one or more of the functions, methods, libraries, or extensions that are new to you in your daily work. Perhaps you will find a more elegant solution with one of these new resources.
3. Use this chapter as a reference the next time you are programming using one of these frameworks. Keep doing so until you no longer need to look at the book because you just know it.

Vulnerability Categories

The first vulnerability that I learned about in depth was SQL injection. I was completely fascinated by the idea that someone could trick my application into doing what they wanted instead of what I told it to do. As a software developer, I felt very powerful. I felt like I could solve almost anyone's problems if I was just creative enough. Finding out that people could take advantage of my code and, by extension, me and my team made me sit up and take notice immediately.

From there, I started learning about individual "hacks:" unique situations where there was a security incident or a data breach, or something else bad happened from a security perspective. Each instance was separate and distinctive to my untrained and inexperienced mind. Over time, though, I realized that many types of vulnerabilities fit into categories and that we could learn about and defend against entire categories, which is more efficient and effective. Creating individual defenses for each specific type of vulnerability is like a whack-a-mole game; you can never get all of them. But if you deploy defenses aimed against entire categories, you can prevent many specific vulnerabilities from happening with one act.

Quite often, people talk about "The OWASP Top 10," which is an incredibly well-known awareness document about the top risks to web applications. People frequently think that those 10 items are the only things they need to worry about, and as a result, they might not plan their defenses in the most successful ways.

As most people have already heard of the OWASP Top 10, and I covered it in my previous book, I decided that when I discussed vulnerabilities this time, I was going to cover them in a brand-new way. I wanted to think outside the box and help you do that, too. You do not need to memorize every single name or type of vulnerability; what you need to understand is the high-level idea of how they work, the risks to your systems and data, and the mitigations needed to defend against them. You do not need to memorize CVE or CWE numbers (which are vulnerability identifiers); as a software developer, you need to understand the basic dangers that your systems face and how to protect your applications from them as best you can.

CVE VERSUS CWE

CVE stands for Common Vulnerability Enumerator, and CWE for Common Weakness Enumerator. CVEs are specific security bugs in operating systems, service packs, libraries, updates, and software that you can purchase, such as Microsoft or Oracle products, and open-source products, such as the Linux Kernel, FreeBSD OS, and OpenSSH. CVEs are in software products that are *not* custom. CWE are patterns of vulnerabilities that you find in *custom software*, software or hardware that is usually built in-house. For example, CVE-2021-44832 is one of the CVE numbers related to the Log4j vulnerability of 2021, which was specific to multiple versions of the Log4j logging library for Java.²⁶⁸ SQL injection, on the other hand, is a category of vulnerability denoted by CWE-89: “Improper Neutralization of Special Elements used in an SQL Command” (“SQL Injection”).²⁶⁹ Although I am not following the CWE notation here, because you can just go to the site if you want to, I am following a similar idea of explaining the patterns (like the CWE) rather than specific instances of vulnerabilities (CVE).

NOTE CWEs are general patterns of vulnerability types; they are not specific vulnerability instances like CVEs.

Design Flaws / Logic Flaws

A *bug* is a problem with the way that you have implemented a solution; it’s an issue with your *code*. A *flaw* is an issue with the *design*; it’s a problem with the way the system was designed. If you have a logic flaw in your design, even if you code it perfectly, you will still have an imperfect end product. There’s no way around it.

When security professionals talk about business logic vulnerabilities, logic flaws, or design flaws, what we mean is a problem with the way that system

was designed from a security perspective, and the problem allows your users to perform unintended behavior(s). Generally, the unintended behavior is potentially damaging; otherwise the security team wouldn't be worried about it.

Here are a few examples of logic flaws:

- The system allows a user to manipulate another user's account, such as uploading data, deleting records, or changing settings they should not have access to.
- The system allows someone to create an account with someone else's email address (when they have no access to that email address). This is why we always confirm that someone is the owner of the email address they sign up with.
- The system allows a user to skip certain parts of a transaction such that they can purchase something without paying or purchase multiple items but only pay for one of them.
- The system allows a user to look up sensitive information they should not be able to, such as seeing other users' data, system data, or internal company data.
- The system allows a user to "use up" resources such that other users are unable to access the system: for instance, allowing search terms that are excessive or create an endless loop.
- The system does not properly log what a user does or allows a user to delete their past actions, such that you are unable to see what the user has done within your system.

How Does This Happen?

Logic flaws happen when developers and designers make flawed assumptions about how users will interact with the application.²⁵⁶ These bad assumptions can lead to inadequate validation of user input and user decisions. Without enough validation of our user's actions, they can escape the flow that you have intended and perform unintended actions within your system.

Logic flaws also happen when we forget to put security controls in the places they need to be. If there is no input validation / output encoding / authentication / access control / encryption / and so on where there should be, you will most certainly have a flaw of some kind.

The Risk

Logic flaws can produce completely unimportant issues, all the way up to critical vulnerabilities. You may have heard people joke, "It's not a bug; it's

an undocumented feature,” to dismiss a logic issue found by a tester, usually because they can’t figure out how to cause damage with it. I would suggest that you fix all logic flaws, even if you don’t see how they can be exploited. Just because your imagination can’t find a way to use the issue for evil doesn’t mean someone else can’t.

The risks of logic flaws are sometimes quite extreme. If the flaw is related to a security control, such as authentication, then your authentication could be misused, and that could have dire consequences.²⁵⁶

Please assume that logic flaws are somewhere between high and critical in nature most of the time. You can downgrade them if you need to on a case-by-case basis.

Prevention

There is no automated tool that is able to find business logic flaws reliably. They might find a few sometimes, but no automated tool that I am aware of at this time is dependable for uncovering this type of issue. This means we need to perform creative manual work, which tends to be a fun part of the job.

There are a few options for finding and preventing logic flaws: performing threat modeling, creating security user stories, doing a review of your design with a security professional to ensure that you have security controls in all the places you should, and, when creating your design, trying hard not to make assumptions of how users will act (they will almost always surprise you). Try to apply as many secure design concepts from this book as you can to all your designs, which will also help to prevent flaws in your applications. Finally, hire a skilled penetration tester, walk them through how the application is supposed to work, and then see what they come up with. They can be very creative (in an evil sort of way).

NOTE It’s a great idea to get advice when making large purchasing decisions, but be careful that the advice you receive is not biased. If you decide to use a service such as Gartner, be aware that many vendors pay to have their products analyzed so that they can be put onto the Gartner lists and in their reports, such as their famous magic quadrant. When hundreds of thousands or even millions of dollars are at stake, that creates bias. Companies like IANS Research do not accept vendor payments to be part of their literature or advice, and it’s part of their mission to provide unbiased technical advice. That said, I work (part-time) at IANS, so I am biased! All that aside, try to get advice from multiple sources when making large decisions such as buying a tool, deciding strategy, and hiring, if possible. Also, if a person giving you advice may be biased, try to take that into account in your decision-making process. And this includes advice from me!

Code Bugs / Implementation Errors

A bug is an error in the way you wrote your code, like a spelling or grammar error. Everyone who writes code also accidentally creates bugs from time to time. It's not on purpose, but it happens! A security bug is like a regular bug, except it has security implications. Usually, this means that it negatively affects the availability, confidentiality, or integrity of the system or its data. In other words, it creates a vulnerability.

This can mean forgetting to put input validation in a place it's supposed to be or coding the input validation incorrectly: for instance, creating a list of known bad characters and trying to block them instead of making an allowlist of good characters and only letting those in. Bad actors can often get around such measures, making it a security bug.

How Does This Happen?

People make mistakes. I know I do, and that's how bugs happen. We forget, we are tired, we are in a rush, or we just don't realize we've made an error in the first place. "To error is human;" that's how bugs happen. Bugs also happen if we don't do multiple types of testing to find bugs. Like running a spelling and grammar checker on an email before we send it, we should always double-check our work.

The Risk

A security bug can create a very small risk or a very large one. Examples include injection, overflows, insecure direct object reference (IDOR), cross-site request forgery (CSRF), and so on.

Not all security bugs are high risk; quite a few aren't even worth fixing. Talk to your security team about the risk they pose until you have enough experience to be able to judge for yourself.

Prevention

To find bugs, we need to do testing, and automated testing really shines in this area. Using a static analysis tool (SAST) will find bugs in the code your team wrote, and a software composition analysis tool (SCA) will find bugs in the code your team did not write (third-party dependencies). I suggest scanning all code with both types of tools, and a dynamic (DAST) or interactive AppSec (IAST) tool if you have one, and then fixing whatever your security team deems important before releasing code to production. Set up automated scans to run regularly if you have the licenses to do it.

Manual code review by an expert can also be a huge help here, as well as a penetration test. But if you don't have the resources for that (experts or the money to hire experts), automated scanning can get you really, really far, especially if you use multiple tools. Your security team can help you decide how much is "enough" assurance for each system under your care.

TIP I shared several free static analysis tools throughout Chapters 6 and 7 that you can run on your code. If you don't have the budget to buy tools to scan your code, there are still many tools (of many different types!) available to you.

TIP Remember, just because an automated tool finds something, that does not mean you have to fix it. Automated tools can find hundreds or even thousands of bugs, and if all your company ever did was fix bugs (instead of creating new features and products), you would likely go out of business. We fix things that cause unacceptable risk to the business so we can keep everyone safe. Talk to your security team and let them help you decide what is worth addressing and what risks to accept.

Overflows and Other Memory Issues

If you program in low-level programming languages such as C and C++, you are forced to deal directly with memory. You need to ensure that you allocate enough of it and that you release it properly when you're done with it, or you end up with a big mess. You need to ensure that you don't overwrite part of your program, the stack, the heap, and much more; there's a lot to deal with when it comes to memory management. Although memory leaks can cause crashes and thus availability issues, the best-known security issues that memory can cause are *overflows*.

Let's go over several different types of memory issues, starting with overflows.

Overflows

A buffer overflow happens when a program assigns more data than it allocated for a buffer or it tries to put data in a memory area past the end of an allocated buffer.²⁵⁷ It's generally an accident. Imagine that someone allocated 20 characters for a string, a user enters 25, the program doesn't validate the size of the input, and then you have an overflow. Writing outside the bounds of a block of allocated memory can result in corrupted data, a program crash, or the execution of malicious code.²⁵⁷

NOTE “Execution of malicious code” means the program exits its intended path into undefined behavior: behavior that a malicious actor is trying to control for themselves. The malicious code is injected into an application by a threat actor. From an application security perspective this is considered the end of the world.

If an attacker is aware that it is possible to overwrite a buffer, they can experiment with it to try to figure out a way to overwrite the stack pointer (the thing that tells the computer where its next instruction is). If they can overwrite the stack pointer, they can fill the buffer with their own shellcode (an exploit, usually written in machine code, which delivers the payload, which is the part of the code that performs the attack) as the next instruction and potentially take over your web server. If you are thinking, “Oh my gosh, that’s terrifying,” then you now understand why memory safety is such a big deal for us AppSec nerds.

Overflow issues can happen with buffers, strings, integers, etc. If you are manually allocating memory to put something in it, an overflow can happen.

Buffer Overreads

Reading unallocated memory can result in sensitive data exposure, and it can happen because someone didn’t validate the input or output of a request, because they messed up their pointers, or due to other memory mismanagement.²⁵⁸ It can also result in an attacker using that information to bypass address space layout randomization, which is the best operating-system-level defense against overflow issues.²⁵⁸

Invalid Page Faults

An *address space* is a range of discrete addresses that correspond to a network host, peripheral device, disk sector, memory cell, or other computer entity.²⁵⁹ If your program tries to go to an address that isn’t listed in the address space, that means there’s no page in memory for it to go to, meaning your program is lost as to where to go next. This is called an *invalid page fault*.²⁵⁸ This is also one of the many ways your application can fall into an unknown state, which you do not want.

When this happens, the operating system will usually pass a segmentation fault to the problematic process, which results in the abnormal termination of the code that made the bad call.²⁵⁸ The operating system can be set up so that attempts to read or write the memory referenced by a null pointer receive an invalid page fault to avoid allowing it to read memory that it should not, but it’s still likely going to cause a crash, throw errors, or have another outcome that you find undesirable. It’s harm reduction, not a solution.

Use After Free

If you have deleted an object, you erase the address of the object after, which removes the pointer.²⁵⁸ If you use that same address to try to access that object after it's been deleted, this can cause issues, and the situation is called *use after free*. This situation can result in your variable being assigned an improperly formed object, which can crash your application as one of many possible bad outcomes.

Uninitialized Variables

In languages that are not memory-safe (such as C and C++), you need to initialize variables (assign a value to them) before you use them. If you have an integer that is not initialized, and you try to add 2 to it, you have a mess. You can't add 2 to a null value. It just doesn't work.

If you try to go to the value of an uninitialized variable, it could crash the system or cause it to fall into an unknown state. This is called *dereferencing a null pointer* (trying to go to an address that doesn't exist).²⁵⁸

When a pointer is used that itself has not been initialized (no value is set for the pointer yet), it is called a *wild pointer*.²⁵⁸ A *dangling pointer* means that the thing the pointer is supposed to point to has been destroyed (an object, for instance), but the pointer itself has not also been destroyed. The pointer points to nowhere. Both of these situations can cause software problems that lead to lack of availability, crashing, and other potential security issues.

Memory Leaks

Memory leaks happen when memory usage is tracked incorrectly or not tracked at all.²⁵⁸ This can result in stack or heap exhaustion (running out of space in either), double free (which can result in freeing an object you did not mean to free), invalid free (sending a bad address, which can corrupt the heap), mismatched free (using the wrong freeing function; it needs to match the way you allocated it originally), or unwanted aliasing (allocating the same memory twice, for different reasons, which can cause confusion or invalid values).²⁵⁸ More simply put, any sort of memory leak can cause a system crash or invalid results, or have your application fall into an unknown state, all of which are terrible from a quality, reliability, and security perspective.

Whenever possible, we want to avoid every single one of these potential memory issues.

ARE RACE CONDITIONS MEMORY ISSUES TOO?

Some people argue over whether a race condition is a memory issue.²⁵⁸ I'm including race conditions as a separate topic here. Whether they are caused by memory or not, they have a separate solution, which means we will give them special attention.

How Does This Happen?

Memory issues happen when we do not use a memory-safe language *and* we mismanage the memory. Using a non-memory-safe language means we need to do more work. *Manual* work. And if we don't do it carefully and then test it thoroughly, we can make mistakes. It's very common.

The Risk

Attackers are primed for memory issues, and we defenders should take them very seriously. Ever since the 1996 article “Smashing the Stack for Fun and Profit” by Aleph One, which described how to use a buffer overflow to attack software, malicious actors have been attacking memory. Besides potential sensitive data exposure, crashing systems, or falling into an unknown state (which can have anywhere from no impact to critical), the threat of remote code execution (RCE) from overflow issues makes memory vulnerabilities a critical risk to any piece of software. RCE is the pinnacle of risk when it comes to application security, as having your systems controlled remotely by an attacker is the most dangerous possible outcome. Please take memory vulnerabilities very seriously; fix them quickly.

Prevention

The top fix for this is using a memory-safe language. If you ever have the chance to push a project away from C/C++ toward Rust or any non-memory-safe to a safe language, please do it. Please push hard, and feel free to quote this book if need be. You will be eliminating a huge amount of risk and effort on your part.

TIP Memory-safe languages are *wonderful*! They do automatic garbage collection; they automatically release and allocate memory for your variables, objects, and everything else you create; and they do all sorts of other helpful things for you to ensure that you almost never have overflow issues (there are rare occurrences, which usually make the news). They are a great help and should always be your first choice over a non-memory-safe language.

If you can't convince the powers that be to switch to a memory-safe language, there's still hope for you and your applications:

- Ensure that your servers all have address space layout randomization (ASLR), Data Execution Protection (DEP), and Structured Exception Handling Overwrite Protection (SEHOP)²⁶⁰ enabled, which guard against overflow-type attacks.
- Monitor memory usage of your systems over time; this can help you detect memory leaks. Set up an alert if significantly more memory is being used or if it creeps higher and higher over time.

- Use safer functions (`strncpy()` versus `strcpy()` or `strncat()` instead of `strcat()`), several of which are defined in Chapter 5 under the C/C++ section of this book.
- Use memory safety libraries to check your code, such as the `Safe C String`, `SafeStr`, and `SafeStringlib`.²⁶¹
- Take secure coding training that focuses on memory issues and/or your language of choice.
- Perform code review, both manual and with a static analysis tool (SAST).
- Use compiler tools such as `StackShield`, `StackGuard`, and `Libsafe`.²⁶¹
- Get a penetration test done or scan your application with a dynamic fuzzer tool.
- Treat compiler warnings as errors. Fix them. Do not ignore them because you can still compile. That's not good enough.²⁶¹
- Turn on warnings (such as GCC) in your compiler, and take the warnings to heart.²⁶¹
- Ensure that your servers are patched and properly hardened regularly.
- Follow the other advice in this book.

INJECTION IS EXPENSIVE

Alice was going over the budget for her department because she was “acting” as the Chief Information Officer while the CIO was on vacation for a few weeks. There was a line item that was *huge*: security incident #67. Alice went to see the CISO immediately to ask what had happened. The CISO explained that they had an incident with injection, which resulted in a data breach; then they were ransomed (which they did not pay, but they *had* involved lawyers. . .), then they had to pay a giant fine, and they were just preparing the message to send to customers. Alice was *astounded*. The cost of the incident, so far, was more than the home she lived in (and Alice had a nice home). And the incident wasn't event over yet! There would still be reputation damage, customer loss, and much more. *Wow*, thought Alice, we had better work harder on preventing these sorts of things.

Injection: Interpreter and Compiler Issues

Injection is when an attacker fools your system into executing their attack code *as if it were part of the system*. It dupes your system into thinking the values they entered into a data field are not data but code. Your system treats data differently than code, but the attacker has misled your applications, so those protections no

longer apply. Your system then interprets or compiles the attacker's code along with yours, and whatever it tells it to do, it does. Against your systems and data!

How Does This Happen?

The injection vulnerability happens because a system has mistaken user-supplied data for part of its own code. This happens because we did not properly validate, sanitize, or escape our input, and we did not use layers of abstraction (an ORM and/or parameterized queries to protect a database, a WAF or RASP to protect a web application, etc.) to protect our systems. We didn't follow the secure coding lessons presented earlier in this book, and we didn't test or review our code appropriately. In summary, we did a bad job.

The Risk

Injection was the number-one entry on the OWASP Top Ten for 20 years,⁷² making it the absolute worst risk to web applications, according to the experts, year over year, until 2021, when it moved down to number 3. When a malicious actor can trick our applications into thinking their code is part of our app, and then their code is compiled or interpreted along with ours, they can perform damage against our users or databases, our operating systems, our host machines, and so on. Any systems that our applications have access to are potential targets, and the attacker's code is granted the same permissions that our application is given. It's utterly horrifying, and the results of such an attack can be staggering.

Injection can result in remote code execution, sensitive data exposure, data integrity issues, availability issues, embarrassment to your organization, deletion of important data, destruction of operating systems, denial of service, and more. Only the attacker's imagination and the permissions assigned to your application limit this type of attack. This is an example of the value that applying the principle of least privilege can provide to our systems.

Prevention

If you follow all the advice in this book, you will never need to worry about injection. But life is not perfect; there are many things at work that we do not control, and sometimes we make mistakes. Let's go over how to avoid injection again because it's imperative that you steer clear:

1. Validate all input to your system against an allowlist. Do the validation on a trusted system. Reject anything that does not measure up.
 - a. If you are using multiple character sets, change them all to the same one (UTF8). Canonicalize the values. Do this before validation and other steps.

2. Sanitize out *or* escape any special characters that you need to accept. Your choice.
3. Use layers of abstraction to further mitigate this possibility.
 - b. Use an ORM database framework to protect your database.
 - c. Use parameterized queries to protect your database.
 - d. Use LIMIT and other SQL controls within queries to prevent mass disclosure of records.⁷²
 - e. Put a Web Application Firewall (WAF) or Runtime Application Self Protection (RASP) in front of your application or APIs for better protection and application redundancy.
 - f. Use an API Gateway to protect your APIs.
4. Code review (can be static analysis).
5. Testing (penetration testing, fuzzing, and/or automated dynamic analysis).

If you find injection in any of your systems, fix it immediately. Do not underestimate this danger.

BLIND SQLi

Bob once had an application with something called “blind SQL injection.” It took him quite a while to understand what the penetration tester was trying to tell him and his team. All Bob saw at first was that the application was giving lots of errors. So what? Once the pen tester explained that she was using an SQL attack to ask true or false questions, and an error meant “no,” and a normal response meant “yes,” his team saw this was not good. But then she explained that she was asking the database, “Is character #1 ‘a’? No? Is it ‘b’?” and so on, until eventually, she got the name of the table, the first field, and what was in it. That was more than enough for Bob to realize that he needed to fix this problem ASAP!

Input Issues

If every input to every application was perfectly validated, with any potentially dangerous characters sanitized or escaped, and all bad input was rejected, I am willing to bet that about half of all software vulnerabilities would disappear overnight. If you take a good look at most of the common weakness enumerators (CWEs), a very large percentage of them would vanish.

When a user, on purpose or by mistake, enters unexpected input into our systems, sometimes our systems behave in unintended ways. This can mean denial of service, data integrity issues, sensitive data exposure, vandalism,

unintended access or privilege, system crashes, legal or compliance issues, or otherwise falling into an unknown state.

How Does This Happen?

Software developers sometimes mistake what is input to a system, forget to do input checking when they should, or do it incorrectly. URL parameters, hidden fields, data from your database that could have been changed by a user, data from an API or another system you do not control, and so on, are all input to your system that sometimes is not validated. When we perform input checking, it should be done on a trusted system: that is, a server (not the front-end JavaScript). We should also do our validation against an allowlist (*not* a blocklist full of bad characters). Unfortunately, software developers around the world make these mistakes all the time, which is why I still have a job.

The Risk

Input issues are the cause or first step of some of the most dangerous types of vulnerabilities, including buffer overflows, injection, cross-site scripting (XSS), and more. Although some input issues are harmless, they can also pose critical risks.

Prevention

Identify and document every type of input to your system, and then ask a colleague to verify you didn't miss any. Walk through the entire system with them if you can. Once you have the list, check them one by one to ensure that you have validated, sanitized, and/or escaped them all properly. Once you've done that, have a colleague review your work or perform static analysis (or both!). Then run an automated fuzzer (usually part of a web proxy or DAST toolset) over every input that you can figure out how to fuzz and fix anything it identifies as problematic. If you can afford it and have time, perform a penetration test.

Authentication and Identity Issues

Network identity refers to the unique identifier or credentials associated with a device, user, or software on a network.²⁶² *Authentication* is the process that computers use to identify devices and users.

When there are problems with authentication or identity, this means the wrong users can gain access to data or systems that they should not.

How Does This Happen?

Having read the earlier chapters in this book, I am sure you noticed just how complex designing and coding a high-quality authentication or identity system can be. On top of that, these systems are stressed, tested, and brute-forced all day long by malicious actors. They are a very attractive target for people with bad intentions, as they are the keys to our digital kingdoms. This type of issue happens because the systems are not properly designed, implemented, and/or tested because they are incredibly complicated and intricate.

The Risk

Having a threat actor gain unintended access to sensitive data, systems, privileges, and so on can pose critical risks, depending on the system. Generally, this type of security risk is always high, never low or informational.

Prevention

First off, buy a system that performs these functions for you, rather than writing your own, whenever possible. If that is not an option, follow the advice in Chapter 2 of this book on this topic. Perform code review and testing whether you buy a system or build one yourself.

THE DEFINITION OF “BUY”

When you hear the expression “Buy, Borrow, Build,” “buy” means “obtain a system that will perform this function.” You do not need to purchase a system if there is a free product (usually open source) available that performs that function at the standard you are looking for. Usually you will need to pay, but not always. Do your research, and you may save your organization some money.

BROKEN BUDGET

Alice went out with a bunch of friends from her executive ladies’ community recently, and one of them told her that instead of doing layoffs, the IT department had decided to cut the budget for tools. As the CISO, this meant her friend was not allowed to renew her contracts for her SAST, DAST, SCA, and other AppSec tools. Although she was happy to keep her staff, she had no idea how they would do their jobs with no tools. The day after the meetup, Alice called her, and they met for a brainstorming session. They managed to find a really good SAST and DAST for free! They weren’t able to find a free SCA that they liked, but they did figure out they could use a free software bill of materials (SBOM) tool and then do internet searches to see if there was anything

seriously wrong with any of the libraries. It wasn't perfect, but when Alice left that evening, her friend was no longer feeling hopeless about her security program. Alice felt grateful for open-source tools!

Authorization and Access Issues

Although authentication and identity issues only happen on first contact with a system, authorization and access issues can happen over and over inside of a system. Every page you go to, every record you read, and every privilege you have needs to be checked *every single time*. You may have access to the main system but not the administration part of the system or access to your own records but not your friends'. Authorization and access control should ensure that you have access to everything you should and nothing more.

When there are issues with either of these (authN or authZ, identity or access control), we end up with unintended access: to software, to data, to networks, to hardware, to anything.

How Does This Happen?

Similar to authentication and identity, not only are these systems complicated, but the repetitive nature of validating every access request can become quite involved or even convoluted from a programming standpoint. The testing of these systems, logging in as every different type of user, and ensuring that you can/cannot access the correct data/menus/images/features/and so on is monotonous at best. This was my least favorite part of penetration testing; the attention to detail required is severe. I am grateful for my pentester friends, who are much more patient than I.

The risk and prevention of these issues are the same as for authentication and identity issues.

Configuration and Implementation Issues

Have you ever gone to log into a Wi-Fi router and realized that the default password was still enabled? The same one everyone else who bought that unit was given? Have you heard about companies with "open S3 buckets" having sensitive data stolen? These are classic examples of configuration or implementation issues; whoever set them up did not follow the hardening guide.

If we misconfigure something or set it up incorrectly, it can lead to devastating consequences. A common example that is part of the latest (2021) OWASP Top

Ten is XML external entities (XXE): XML-enabled servers that accept DTD files when they should not, creating a vulnerability. To fix this issue, all you have to do is disable that feature, but *you need to know the problem exists* in order to fix it. Generally, the administrators or programmers of systems that have configuration or implementation issues are not aware they have a problem until it is too late.

How Does This Happen?

Mostly these issues are caused by one of two things: not following the hardening guide or a complete lack of awareness that a hardening guide exists or should be asked for and then followed. Most organizations that have XXE aren't aware of what XXE is, that it poses a threat, or that they are affected. If they did know, with such a simple fix, most people would address it.

The Risk

Security misconfiguration can be anywhere from informational/low to critical and should be evaluated on a case-by-case basis.

Prevention

You may think that avoiding configuration mistakes is a matter of willpower, but there are several steps you can take to ensure that you never fall into this trap, meaning you don't have to depend only on your memory:

1. Follow all the best practices you have learned in this book.
2. When you set up a system, follow the hardening guide. If it did not come with one, ask for it.
3. Once a year, perform an audit of all systems to ensure that they are still following the hardening guide. Check if there's an updated guide.
4. Keep all systems patched and updated.
5. Have a versatile, fast, and efficient patching and update process.
6. Have a change management process that documents changes. Ideally, it should be automated and painless.
7. If possible, perform testing against important or all systems, especially those that are mission critical, integrate with mission-critical systems, or touch sensitive data.

NOTE When deploying a security product, ask for feedback from those who are affected or have to use it. One of the places I was contracting deployed a tool that was supposed to remove admin rights, but they misconfigured it, and the result was that no

one could access localhost. The developers complained, and the security people, who had never been programmers, had no idea just how important being able to compile and run code locally was for programmers. Localhost is indispensable! I got involved, we fixed it, redeployed it, and it worked as intended (no more admin rights, but localhost functioned again). The programmers rejoiced. When developers (or any of your users) complain, listen! We never want a security tool to make the rest of IT miserable.

Fraudulent Transactions

I have had my credit card stolen at least five times in my life. Once the physical card was taken from my wallet by someone I worked with, who made one large purchase at a nearby liquor store and then threw it out. The other four times, it was stolen online, and the thief performed *several* fraudulent transactions. I was relieved that the credit card companies did not charge me for any of the false purchases, and the only suffering on my part was waiting for the new card in the mail. But not all fraudulent transactions are so obvious or pain free.

Transaction fraud is any deceptive activity intended to acquire money, goods, or services during a financial transaction without paying for them.²⁶³ It's a risk involved not only with money but also with credits for used books, points within a video game, issuing a ticket to a concert, or any other situation that involves the trading of a thing or service for something else.

How Does This Happen?

People have been stealing from each other since the beginning of time. Some people, when they see a way to take from others or a faceless corporation, jump at the chance. It's our job to protect our systems and data from such people. When we do not roll back and fail closed after a transaction that has an error, there's a chance it could cause a vulnerability. If our application has a race condition (more on that later), improperly validated input, logic issues, and so on, fraud can happen.

The Risk

If someone is cheating at an online game, you might think there is little risk involved, but what if there's a huge bet on the outcome of the game? If someone figures out how to use the same coupon 300 times instead of once, perhaps it is enough to put a small company out of business. Constant fraudulent transactions in the financial industry are the main reason we all pay so many fees and so much interest.

Online payment fraud alone cost the world \$48 billion in 2023.²⁶⁴ With 8.1 billion people alive in 2023, that's \$6 per person for one single type of fraud. There are several other types. People lose their retirement funds, their online identities, their homes, and more from transaction fraud. The risk is potentially catastrophic on an individual or organizational level.

Prevention

This type of vulnerability can be caused by an implementation issue (coding) or a logic flaw (design). This means that to prevent this type of issue, we need to perform threat modeling and design review, as well as careful code review on each transaction control in our systems. We need to ensure that we *always roll back and fail closed* if there's an error during a transaction (as per previous advice in this book). I would also suggest that if the stakes are serious, you consider a bug bounty program or the involvement of multiple security researchers to perform extremely extensive testing of such features.

IDENTITY THEFT

Two of my four parents were in the data breach from the Canada Revenue Agency in 2021. As a result, my parents need to have credit monitoring for the rest of their lives in case someone tries to steal their identities. I don't know about you, but as an IT security professional it makes me feel ashamed that the people I care about most are not properly protected by our industry. Although I can't go back in time and prevent that breach, I hope that this book can help my readers and me work toward a much safer future for everyone.

Replay Attacks

A *replay attack* is when a threat actor captures transaction, authentication, or access control information in transit and then retransmits (replays) the information with the intent of producing unauthorized access or an unauthorized transaction.²⁶⁵ Someone gets a copy of the request as it travels over a network and then sends it again and hopes they can reproduce the same results. This can happen with buying groceries online, transferring money from one account to another, or any other type of digital transaction.

How Does This Happen?

If we do not have a per-transaction random value (or other type of token), a one-time password, or a timestamp that we validate when we receive requests, and/or we don't validate for duplicate transactions, this type of attack can happen.

The Risk

The risk of a replay attack is similar to online fraud; some are tiny, some are huge. That said, fraudulent transactions have more possible vectors of attack than replays and thus, are more common.

Prevention

To prevent a replay attack, you can do one or more of the following:²⁶⁶

1. Send your session ID along with a random value (nonce) and validate both on the server side before processing any transactions.
2. Do the same as item 1 but with a one-time password instead of a random value.
3. Same as item 1, but with a timestamp for every transaction request, and then validate that it arrives within a certain amount of time from the timestamp. This does not eliminate the threat; it just reduces the time window.
4. Same as item 1, but also send a timestamp. Record the timestamp along with each new transaction request. Compare new transactions for duplicates, and if the timestamps match, then block.

Note: there are even more advanced ways to protect against this type of attack, such as the Kerberos protocol, AODV protocol, and Challenge-Handshake Authentication Protocol, but I feel that diving into these network-based protocol protections is beyond the scope of this book.²⁶⁶

Crossing Trust Boundaries

When we accept input into our application from an untrusted source, it is considered untrusted or “tainted.” Wherever we got it from is called the *source*, and the first time we use that value within our application is called the *sink*. We should always validate, sanitize, and/or escape a value to prove that it can be trusted (or reject it) *before* we sink a value. The value is considered “tainted” until we have proved it is trusted.

FLOW ANALYSIS EXAMPLE

Imagine that you are using the search feature in a social media platform such as X or BlueSky. You enter “#infosecgardening” into the search field. That value is tainted or untrusted when you enter it. Then the application should perform input validation, as well as potentially sanitization or escaping of any special characters it needs to accept. Assuming the value passes these hurdles, it is considered trusted or no longer “tainted.” The application then is ready to “sink” the value, which means use it for the

first time. It queries the databases for posts with that hashtag. The application proceeds to show you posts with beautiful images of security professionals' gardening efforts. The world is a safe place again.

When we perform static analysis (SAST) or manual secure code review, we document every input as trusted or untrusted (a source), and then we track that value until it is either validated or sunk (used for the first time in our code). If it is sunk *before* it is validated, we will have a security issue. It might be injection, XSS, or something else, but there will most certainly be a problem from a security perspective. In static analysis, this process is called *flow analysis*, and both first- and second-generation SAST tools perform this as part of their functionality.

TIP Some SAST tools can perform flow analysis across multiple files (inter-file and cross-file), whereas others do not, which is something to look for when shopping for a tool. You can miss some sources/sinks if you cannot check across multiple files (for instance, from the front-end JavaScript into the backend code on the server).

When we use a user-supplied data value before establishing that it is trusted, this is called *crossing trust boundaries*.²⁶⁷ We *never* want to cross trust boundaries.

How Does This Happen?

Missing or improperly implemented input validation is the bane of every AppSec professional's existence. It's almost never intentional; it's just human error.

The Risk

The risk of crossing trust boundaries can be low or critical. Example outcomes include poorly formatted query requests with incorrect results (low) all the way up to SQL injection that results in RCE (critical). Each instance must be evaluated on a case-by-case basis, but I recommend always fixing this issue if you find it.

Prevention

In an attempt not to sound like a broken record, please read and follow the advice in Chapter 2 of this book, in the input validation section. You should be well prepared by this point to avoid this issue.

File Handling Issues

If your application handles files improperly, it can result in the application crashing, the file attacking your application or server, or ruining other data you

had saved in the same place as you put that file. The code that is supposed to process the file may have loopholes or flaws, it may not validate the type of file or other attributes correctly, or it may not scan it for malware.²⁷⁰

The file could be from a user, a partner organization, or within your own company, included in your applications, edited by your applications, uploaded and saved into your database, and so on. Any poor handling of a file can create a vulnerability in this category.

WARNING Allowing unauthenticated members of the public to upload files to your application is generally considered an extremely high-risk activity. Please avoid it if at all possible.

How Does This Happen?

Improper file handling could be caused by either a bug or a flaw, a coding error, or a design issue. Often, we do not check the file headers to ensure that they reflect the same type that the file extension says; the name or path of the file could be an issue, or there could be malware inside the file. There are many ways that this could go badly if we are not very careful.

The Risk

An attacker could potentially access sensitive data, run operating system commands, spread a virus across our network or data, or crash our application. Attackers could include remote (RFI) or local files (LFI) into your applications, download files or folders from your servers, upload unintended, dangerous files, or edit files they should not. This situation also falls under the category of improper input validation. The risk can be anywhere from medium to critical, depending upon the exposure.

This issue could negatively affect the confidentiality, integrity, and availability of the system, the system's data, and your entire network.

Prevention

Read Chapter 3 “File Uploads” section of this book.

TIP Consider using the free open-source tool AssemblyLine from the Canadian government to check your files for malware. Thanks, Canada!

Object Handling Issues

I want to start this section with a short review of object-oriented programming (OOP). OOP has several features that separate it from declarative (imperative) and functional programming, which can either help or hinder the reliability and security of an application, depending upon design and implementation. Let's explore each one.

First, a surprisingly succinct definition of OOP:

An encapsulated unit of data and behavior.²⁷¹

—Reddit User mcr1974

Prominent Features of OOP

Inheritance is the ability to create a new child class from a parent class and extend it with additional data fields, override the inherited classes to change their behavior, or add new classes altogether. This means you can build security into the parent class and then improve on it or ruin it with the inheritances you pass down to its child classes.²⁷²

A class includes data structures and methods, which you can then create an instance of, which is called an object. Class -> Object (Instance).²⁷³ A child class can inherit the characteristics of a parent class, which it can adjust, override, or add onto, just like a human child does to its parents' genes. The child class is based on the parent class, but it is still its own distinct entity.

Imagine that you have a vehicle class. Each vehicle class has wheels, a steering wheel, a dashboard, and a windshield. A child class of vehicle could be a motorcycle class, which specifies that it only has two or three wheels and that the driver is not situated inside the vehicle. Another child class could be a car class, which states that it always has four wheels, and the driver and other occupants are always situated (protected) inside the car. Each type of vehicle (car, truck, jeep, tank, ATV, etc.) *inherits* from the vehicle class.

Back to the car class. Imagine that I create two instances of it: a Leaf object and a Fit object. I currently own a Nissan *Leaf*, and I previously owned a Honda *Fit*. Although I would never recommend specific cars to anyone, they are the ones I am most familiar with, so we will work with those examples going forward.

Encapsulation hides the internal details of an object from the rest of your system. It means that only the object itself can access and change its own data, whereas other objects or classes can only interact with it through its public methods. This helps maintain data integrity, avoids unintended changes, and prevents

unauthorized access. We use *access modifiers* (private, protected, or public) to define who and how your data members and methods can be accessed.²⁷²

Back to the car example. Last year I bought my first-ever brand-new car in my life, an electric Nissan Leaf. My previous car was a used Honda Fit. I could do anything I wanted to my little Honda. I could run a scanner on it that I bought on the internet, figure out what the flashing light on my dash meant, and theoretically fix it (although generally I got someone else to do it). With my Honda, it was like a structured program; I could access any part of it I wanted to, for better or worse (I only broke something once!). But my new Leaf is more like an OOP program, where almost all of the methods and variables are marked private or protected. I cannot perform a fluid change, jailbreak my operating system, or almost any other maintenance except automated software updates without voiding the warranty. I'm locked out. The internal functionality of the Leaf car object is *encapsulated*.

Polymorphism allows your program to access objects of different types through the same interface. The same thing happens but in different forms.²⁷⁴ Personally, I find this definition extremely confusing, so let's break it down.

NOTE *Polymorphism, an ancient Greek word, means "many forms."*

Imagine that you could have a method or operator behave differently depending on the context (the object calling it). When a parent class reference is used to refer to a child class object, this is a common use of polymorphism.

It's like when my mom says, "MS. JANCA, DID YOU FORGET TO PUT YOUR DISHES IN THE DISHWASHER?" But normally, she calls me Tanya.

Examples of polymorphism include

- *Overloading methods* within an object by creating several methods with the same name but with different inputs (parameters) or outputs (return value). There must be a difference in one of those characteristics (input or output) in order for it to work.
- *Overriding a method* from the inherited superclass (parent class) by redefining the class with the same name, inputs, and outputs but with different behavior.
- *Overloading an operator* to create new types, redefine built-in operators to work with the new types, or perform the same and similar behaviors but against an object class.

Back to my beloved vehicles. The Fit had a regular car combustion engine that used gasoline to get me around. The Leaf has an electric engine that works very differently, but it also gets me around. It's still called an engine, but it's completely distinct in how it works (methods). The Leaf car object is powered

by electricity and a battery, so it has different inputs, but the output is the same (Tanya gets from point A to point B). This is a loose example of polymorphism.

Abstraction means reducing complexity by hiding unnecessary object details from the user.

Every car has abstraction, but some more than others. To drive a car, you don't have to know how to build an engine, but you should definitely take it to the shop if the engine light comes on and you don't know how to check what it means for yourself. I remember my cousin showing me the "engine" of her Tesla: it was just a trunk at the front of the car. She keeps her purse there. The car still runs, but the engine is completely abstracted away from the user.

But back to mishandling objects.

Deserialization and Other Object Handling Issues

There are many ways to mishandle an object, but the two best-known are insecure deserialization and broken object level access control (BOLA). Accepting a serialized object from an untrusted source and then deserializing it creates a huge risk, and BOLA can mean sensitive data exposure; negative effects on the availability, integrity, and confidentiality of your system and/or data; or escalation of privilege.

How Does This Happen?

OOP is complicated. Getting abstraction, encapsulation, and the rest of it "right" is hard. Mishandling an object can be an implementation or design issue, as it's especially easy to make a mistake when doing inheritance or polymorphism, which means some of the names are the same but they do different things. When complexity is high, it is easy to make errors if we do not have checks and balances.

The Risk

The risk of insecure deserialization is high, and so is the risk of BOLA. Other object mishandling is anywhere from medium to high risk. Any object mishandling will be above low, as your program has the risk of crashing.

Prevention

There are many things we can do to reduce or remove the risk of mishandling an object. For starters:

- Apply the concept of least privilege to objects. This means carefully evaluating the permissions you give variables and methods and how you use and call the objects.

- Never deserialize an object from an untrusted source.
- Write out the permissions, inheritance, polymorphism, abstraction, and encapsulation an object will have, and then have someone review it. Walk them through it, talk it out, and draw it if it helps. I find that explaining something to someone else helps me work the kinks out of almost any plan. Document it.
- Threat model your design, and include your objects in the discussion.
- Do a design review with the security team. You can combine the effort with your threat model if that works for you.
- Do code review, especially around complex object implementations.
- Use reflection and introspection (complex object implementations) sparingly and cautiously, and restrict their access to trusted or authorized code.²⁷²
- Use a memory-safe language if doing OOP. If you must use a language that is not memory-safe, follow the advice earlier in this chapter on memory issues.
- Avoid pointer arithmetic and smart pointers. Ensure you check the bounds and validity of pointers.²⁷²
- Ensure that you properly initialize objects after instantiating them using your constructor.²⁷²
- Destroy objects after you're done with them using your destructor. Garbage collection should be part of this practice.
- Consider using design patterns that are considered useful for data security, such as Singleton, Factory, Proxy, Strategy, or Decorator, if they make sense for what you are building.
- Do the rest of your secure system development life cycle (SDLC), and use all of your automated security testing tools. It won't be perfect, but it will help.
- Consider a penetration test.

Secrets Management Issues

Secrets, as discussed previously, are private values used by computers to authenticate to each other, like a human uses a password. A secret can be a hash, a certificate, an API key, a password, and so on. When we say a secret has been *mismanaged*, we usually mean it got into the wrong hands (or we suspect that it has). It was spilled onto the internet, a malicious actor got a hold of it, or it was otherwise exposed accidentally. Just because a secret is spilled doesn't

necessarily mean someone will use it against you, but you need to react as though it will happen.

How Does This Happen?

Secrets used to be stored in code on purpose. We now know that is not a good plan. There are tons of secrets in code thanks to those legacy lessons, but also because people make mistakes. They forget. They don't have a secret management tool, and they aren't sure where to put their secret. Or, maybe they don't realize it's a secret, such as a license key, so they store it insecurely. Sometimes you keep the secrets in your code, but only in the dev environment, and forget to remove them before you go to prod. There are many ways this can happen, and it happens a lot.

The Risk

Besides the obvious sensitive data exposure (the secret itself is sensitive data), generally this type of issue results in stolen money, data, or other resources, privilege escalation, access control issues, and so on. In the worst case, it can lead to remote code execution. If a live secret has been spilled, you should consider it a critical risk, rotate the secret immediately, and initiate the incident response process to find out if you've been compromised.

Prevention

This is a multistep process. First, you want to find all the secrets that are stored insecurely in your code with a secret scanning tool. There are paid tools and free tools, and the main difference is that free tools will miss more stuff and have more false positives, so they take more time. Paid ones give better results and thus, take less time, but they cost money. Choose accordingly.

Once you have found all your secrets, you need to figure out where to store them securely. The best place to store them is a secret management tool, but if you don't have one of those, there are other places you can store them (for instance, in environment variables if you are using Python, depending on your configuration and setup). Environment variables are not always a safe place; you will need to do some research if you decide not to use a secret management tool. If you do not have a secret management tool, I suggest you contact the security team and then (1) ask them for one or (2) ask where you should be storing them, if not in your code. Follow their advice.

TIP Ensure that your secret management tool's access control is locked down! It literally contains the keys to your (digital) kingdom. Be very strict, follow the hardening guide, and apply least privilege.

Once you know where to store your newly found secrets, you will need to do a few things:

1. Rotate each secret, one at a time. This means resetting the password, API key, and so on. Have the system generate a new one. Ensure that it is random and unique. Never reuse an old secret.
2. Put the new secret wherever you are supposed to be storing secrets from now on.
3. Change your application so that it gets the secret from the new place.
4. (optional) Configure your CI/CD to get the secret from the new location as it builds your app.
5. Set up a precommit hook on your version control/code repo so that it scans with your secret scanner each time you check in your code. Make it block the check-in if it suspects there's a secret in the code. This will ensure that you never have to repeat this process again, *and* it will prevent security incidents. #win

Race Conditions and Timing Issues

We already covered race conditions in *Alice and Bob Learn Application Security*, but it's important, so we'll cover it again. A race condition occurs when two computer program processes, or threads, attempt to access *the same resource or process at the same time* and it causes a problem (potentially a vulnerability).²⁷⁵ Race conditions are more common with multithreaded applications.

Timing issues are generally intertwined with race conditions. Although an attacker can launch tests and attacks to find out how long a process takes (called a *timing attack*) so they can use this information as part of a larger attack, what we will discuss here is when the timing of a process or request interferes with another, leading to a race condition. Although you do not want attackers timing each of your requests to see which ones can cause problems for you, the concern here is much larger and usually leads directly to bad outcomes.

How Does This Happen?

Any situation where more than one system is trying to use a resource or process at the same time can cause a race condition. If you were to try to withdraw money from a bank account using two different users at the same time, this could cause a race condition. Imagine if both people withdrew the same money. Or, your account went into the negative. Or, the entire thing froze and then crashed. All of those outcomes would be terrible.

Imagine two processes racing to get to a resource first. That is how this category of vulnerability got its name. If we do not lock a resource before we use it, race conditions can happen. Locking a resource while we use it, and then unlocking it after, prevents potential race conditions.

We can also run into race conditions if we are not managing state correctly and safely.²⁷⁶ If we have various external processes that set changes to the state in progress, but they are not properly validated or interrupted, or these checks are not considered part of design specifications, this could lead to a race condition.

If our programs include asynchronous operations, and those operations are used to manage shared resources, this could also create a race condition.²⁷⁶

The Risk

The outcome of a race condition is unpredictable. It could result in anything from denial of service for one user or many to incorrect accounting, system crashes, privilege escalation, data corruption, or even a data breach. Commonly, a race condition allows you to exceed a limit within the business logic of your application (called a *limit overrun race condition*), and that's what defines it as a vulnerability instead of just a programming bug.²⁷⁷ The issues caused by race conditions aren't always obvious, so at first they might seem benign, but an application acting in an unpredictable way is never good. You *must* fix race conditions.

Prevention

To avoid race conditions, we must do a few things, most of which are testing and design considerations.

First, when designing systems, ensure the following:

- Limit complexity whenever possible. Complexity often leads to problems. Simplify whenever possible.
- If your design includes shared resources, ensure that if they are being accessed and modified by multiple threads or processes concurrently, you review this part of the design with a security expert, then review the code when it's complete, and then test this part of the application very carefully.²⁷⁶
- Follow the same steps as the previous bullet point for any code that handles state management and any asynchronous operations. These are also areas in your system that could cause race conditions.²⁷⁶
- Design thorough locking functionality for shared resources and processes.
- Use datastore integrity and consistency features such as column uniqueness constraints, only using a single stored procedure (as opposed to

multiple) to manipulate a single system value (such as validating a financial transaction before it completes), and using concurrency features in your database.²⁷⁷

- Validate transactions with a second/separate check before allowing them to complete.
- Ensure that your session handling framework manages sessions in a consistent manner.²⁷⁷ If you update all the session variables together in a batch instead of one at a time, there cannot be a race condition for those variables. Strive for consistency whenever possible, even if it's slightly less efficient in this case.
- If you decide to manage your session state on the client side, ensure that you encrypt it and are aware of the risks involved in doing so.²⁷⁷
- Consider avoiding the use of shared states.²⁷⁷
 - Instead, your threads could pass messages directly to each other, which are saved into private memory for each process, meaning no chance of race conditions for those threads.
 - You could also use immutable objects, which cannot be changed once created, which would eliminate the possibility of a race condition.

When coding:

- Use a more modern programming language and the latest and greatest version of your (modern!) programming framework. They include features that will help ensure synchronization.²⁷⁸
- Use mutexes, semaphores, and monitors to synchronize your threads.²⁷⁸
- Use thread-safe data structures and synchronization primitives provided by your programming language or framework.²⁷⁶
- Use atomic operations (operations that are instantaneous from a programming perspective) for critical sections of code to prevent interference from concurrent threads.²⁷⁶ An example of when to use an atomic operation would be when your program changes state.
- Write code to validate all input and any changes before your application transitions to a new state.²⁷⁶
- If there is no functionality available on your framework for handling the previous advice, get a library that can (but check it for vulnerabilities first!).²⁷⁶

When testing:

- Perform all the normal tests, such as static and dynamic analysis, but also stress and performance testing.

- When engaging a penetration tester, inform them that you are doing multithreading, managing state in an unusual way, or have concerns about your locking mechanisms. Ask them to pay specific attention to any timing issues or potential race conditions.
- Perform concurrency testing. If you know you're using multithreading or have the potential for race conditions, test for them specifically. This can be performed by your team, the security team, or a consultant.
- Monitoring for race conditions via log analysis, network traffic analysis, and system performance monitoring. Look for unusual patterns that may indicate a race condition.²⁷⁷

As you can likely see, race conditions are complicated! They can cause quite a bit of damage and are easy to miss if we are not careful, as we are creating complicated systems. Please be wary and avoid these issues whenever possible.

Resource Issues

Resource issues, from a security perspective, mean that a resource is improperly released, shut down, or made otherwise unavailable in error. Your application needs to use a resource, but it can't because the resource is accidentally tied up somewhere else.

Many years ago, I programmed an application with recursion, and I opened a database connection *inside* the recursive function. The application crashed after about 10 minutes of being in production. I opened connection after connection until it exhausted the database server and crashed. This was the biggest programming mistake I ever made in my career (in my opinion), and I was incredibly embarrassed when I figured out what I had done. Oops!

NOTE Race conditions also fall under this category of resource issues, but we are going to skip that part because we just covered it in the previous section.

How Does This Happen?

If we do not actively check to ensure that we have released every resource after we are done with it, this can happen. If you are not using a memory-safe language, this is a very easy way to fall into this trap, as you are forced to manage memory on top of your session, connections, and everything else on your (programming) plate.

The Risk

Most unreleased resource issues result in availability and/or reliability problems, but they could also cause a denial of service or sensitive data exposure in certain situations.²⁷⁹ That's the *A* in CIA through and through, which is often the most important to businesses when it comes to the confidentiality, integrity, and availability (CIA) of their systems and data. This means it's usually considered a serious risk.

Prevention

To prevent unreleased resource issues, start by using a memory-safe language, plus ensure that it's a modern programming language and framework. That will solve unreleased memory for you most of the time.

Keep track of anything you open or allocate, such as memory, file space, objects, connections, and so on. Then validate that you've closed or released everything when you are done using it (including every part of an object, such as its member components).²⁷⁹ Make a list, and check them off. That makes it easier to ensure that you have everything.

You should also use any features in your framework that can help you in this regard, such as the `using` statement in VB and C#.NET, which can be used to open connections to a database. It forces you to close the connection, or it will not compile, unlike the `open()` function, which will absolutely allow you to open a connection and not close it, which can result in a giant production crash. Sigh; I might have learned this one the hard way.

Falling into an Unknown State

Programs are sets of instructions for computers to follow. If the computer can't find its next instruction, which instruction is next is ambiguous, the instruction pointer is dangling, or one of many other situations, the computer doesn't know what to do next. When the computer isn't sure what to do next, it *falls into an unknown state*, meaning we, the programmer, have no idea what it will do next, and neither does the computer. The results of such a situation can be catastrophic.

NOTE Some people use the term *undefined state*, whereas I choose to use the term *unknown state*. Use whichever one makes the most sense to you.

Many security vulnerabilities happen when we fall into an unknown state, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. The

list goes on. Literally almost anything can happen when the computer doesn't know what to do next, including nothing at all.

How Does This Happen?

Falling into an unknown state happens when the computer doesn't know what to do next, and it takes a guess. As computer programmers, we do not want the computer to decide on its own what to do next; we want it to follow our instructions (our code). When the computer is guessing what to do next, almost anything (that a computer is capable of doing) can happen.

The lower the level of the language, the easier it is for this to happen. If we are programming in C/C++ and using pointers, it's easy to mess things up. If we're using something with a lot of abstraction, such as .NET C#, it's harder but still very possible.

NOTE Our program must always know what to do next.

The Risk

Almost every category of vulnerability can be caused, made worse, or initiated by falling into an unknown state. Almost anything can result from this situation, and it is never good. If your program is not doing what is expected, it is always bad for business. You should consider this situation critical, and this type of bug needs to be fixed immediately.

Prevention

I would say that of every type of vulnerability out there, falling into an unknown state makes dynamic, penetration, and manual testing the most worthwhile. Although you might be able to spot such issues with threat modeling, static analysis, or manual code review, interacting with the application must always be the final verification that it is safe from this type of issue.

You should always ensure that you use `else`, `otherwise`, a global exception handler, `try/catch`, and anything else provided in your language or framework to “catch” unexpected issues, and then give your program a new direction/instruction. Do not catch an error and then do nothing about it; that does not solve the problem. You must give your program new instructions.

If you follow the rest of the advice in this book, including using a secure SDLC when you create software, you should be in a great place to prevent this type of issue.

Chapter Exercises

You can find the answers to these questions in Appendix B, “Answer Keys.”

1. Of all the categories of vulnerabilities in this chapter, choose your favorite.
 - a. Search online to find a real-world example of this vulnerability, and then read about it.
 - b. Tell a friend or colleague what you learned.
2. In your own words, explain what a business logic flaw is and why automated tools can't find them reliably.
3. Give two examples of fraudulent transactions. Explain how each of them could negatively affect the victim. Your examples do not need to include currency or financial fraud.
4. Give two examples of configuration or implementation issues (for any technology system of your choice). Explain how each of them could negatively affect the system that is wrongly configured or implemented. What could be the outcome?
5. What's the difference between authentication and authorization? Can you have one without the other? If so, why? If not, why not?
6. Why are secrets so difficult to manage? If you disagree, say why. If you agree, explain.
7. Of all the categories of vulnerabilities in this chapter, which seems the worst? Or scariest? Why? Explain in your own words.
8. Of all the advice in this chapter for avoiding these issues, which do you plan to implement? Name one or two and why you want to do them first.



Summary of Part II

In this section, we covered technology-specific advice, popular languages, frameworks, and vulnerabilities. Here, I will attempt to summarize this huge amount of information for you in a meaningful way.

Checklist of Technology-Specific Secure Coding Advice

Although APIs, mobile apps, web sockets, serverless, and IoT are all different technologies, there are certain elements that are similar for all. Use this high-level checklist as a starting point for any new technology:

- Create a standard and/or written guidance for how to secure each technology your organization uses (please use information from this book as a starting point), and then teach it to everyone in your organization who is affected.
- Build protection into your systems from automated attacks such as scripts and bots (or buy a system to handle this responsibility).
- Add layers of abstraction and multiple defenses whenever possible.
- Perform an inventory exercise every year, or buy a tool to automate this if possible.

- Be very careful with authorization (access control) on every system, at every level. Every single function call, page view, record returned, and so on: check before you grant access.
- Add rate limiting and other limits at every possible opportunity. Nothing should ever be permitted to be “unlimited.”
- The systems you create and maintain should use service accounts (not an individual’s account) to operate on a network.
- Ensure that your data is encrypted in transit and at rest (including local storage) and that your secrets are managed properly (not stored in your code).
- Decommission old devices/systems carefully and intentionally.
- Apply least privilege and other secure design concepts at every chance.
- Obfuscate any code that may fall into the wrong hands.
- Offer MFA to your users if possible, and apply strategies to avoid credential stuffing and brute-force attacks.
- Remember to respect your users’ privacy.
- Plan for updates that your users will be able to apply easily if they are expected to perform that action themselves.
- Provide a hardening guide to the end user, if applicable.
- Set all defaults in your system to secure defaults. Make the user perform extra work to disable them.
- If your system is physical (hardware/device), include physical security features (not covered in this book). Securing only the software is not enough.
- Follow the advice in this book, including using all the available security features available in your framework, protocol, or language of choice.
- Follow a secure system development life cycle, including logging, monitoring, and alerting, all the way until the system is decommissioned.

Checklist of Secure Coding Advice for Languages and Frameworks

Although every programming language and framework is different, there are some threads in common when it comes to security. Use this high-level checklist as a starting point for this part of the book if you are not ready to implement the in-depth advice contained within Chapters 6 and 7:

- If writing an application from scratch, please try to select your language and framework based on how many security features are available and

how modern they are. Newer frameworks and languages are usually safer by default. Use all of the security features provided.

- Always choose a memory-safe language over one that does not provide memory safety by default.
- Use your language and framework as intended. Do not undo the security they provide out of the box. For instance, do not write inline JavaScript.
- Be extremely careful with anything to be reflected to the screen, to avoid cross-site scripting (XSS). A very large percentage of these two chapters are dedicated to that one vulnerability.
- Use language-specific tools such as static analysis (SAST) and linters to improve your code whenever possible.
- Ensure the integrity of any scripts, plugins, or anything else you download/add to your code via code signing or another method.
- Make a list of bad functions (as provided in this book) to avoid, and then either grep through your code occasionally to make sure you're not using them or make a rule in your SAST or linter to find them and report it when you've used one in error (this is one way to create a "secure guardrail").
- Never set a boundary to *, such as CORS, an origin, or anything else that was designed to restrict.
- Do not load code from untrusted sources. This means style sheets from random places, JavaScript from a site you do not know, or running scripts against your database (even if it's from a customer you really like).
- Use the latest version of the language and framework, and do your best to keep them up to date. At the very least, stay on a supported version.
- If you find a reproducible security bug in your language or framework, report it to the framework maintainers so that everyone can benefit from the fix.
- Remember that insecure deserialization can happen in any language that allows serialization, not just Java. Do not accept serialized objects from untrusted sources; if you must send one yourself across a network, encrypt it first with a private key, and then decrypt it on the other end to ensure its integrity before deserializing it.
- For SQL or any other query language, harden your server and use parameterized queries.
- Use the logging features in your framework, and protect your logs.
- Do not use production data in nonproduction environments. Please!
- Do not run anything as root in production. We know better.
- If the creators of your language or framework provide a security guide, get to know it well.

- Whenever possible, use immutable data types, classes, output types/values, and so on.
- If your language has a “strict” mode, turn it on.
- Use an object relational mapping (ORM) framework with your database, if available.
- If your framework provides cross-site request forgery (CSRF) protection, turn it on. It is not always enabled by default, and it may require effort on your part.
- If you need to manage memory manually, do it intentionally and do it the same way for every program your organization creates/maintains. It will make it much easier to spot memory issues if you all handle them the same way.
- Always initialize your variables. Do not leave it to chance.
- Treat compiler warnings like errors and fix them.

Summary of Vulnerability Issues to Watch For

You do not need to memorize every type of vulnerability to be a great programmer, but understanding the risks and how vulnerabilities work can prove helpful:

- Logic flaws cannot be picked up easily by automated tools; you will need to find them via threat modeling, security user stories, and/or manual security testing (penetration testing). These issues fall into this category: authentication, authorization, fraudulent transactions, race conditions, and replay attacks.
- Coding and implementation bugs can often be found with automated tools, such as static or dynamic analysis, or manual processes, such as code review. Due to these issues being so easy to find, it is extremely important that you fix them quickly. This category includes injection, memory issues, input issues, configuration issues, and crossing trust boundaries.
- Some issues can fall into either a design flaw or an implementation issue (bug), including misconfiguration, file handling issues, object handling issues, poor secret management, resource issues, and falling into an unknown state.
- Authorization (access control) issues are more common now than ever before. You must be extremely diligent in validating and revalidating access, always denying by default to avoid this issue.

- Problematic secret management is extremely common, and malicious actors are very well prepared to exploit any weakness in this area. Spending time to clean up secrets and implement prevention mechanisms is time well spent.

After reading this summary, I hope that you agree that we must focus on both the implementation and the design of systems to create truly rugged, tough, reliable systems that serve our customers well—which is what we all want.

Now, let's proceed to the third part of the book, which covers the processes that will ensure you create excellent software and a secure system development life cycle (S-SDLC).

Part



Secure System Development Life Cycle

In This Part

Chapter 9: Requirements
Chapter 10: Design
Chapter 11: Coding
Chapter 12: Testing
Chapter 13: Release/Deployment
Chapter 14: Maintenance
Chapter 15: Conclusion

Requirements

No matter what type of project you are starting, you need to know what you are building. Whether you are following the Agile, Waterfall, or DevOps software development methodology to build software, you still need clear direction (in writing) to know what the final product will be. In this chapter, we're going to cover what you and your security team can do as part of the requirements phase of a project to ensure the security of the system you are creating.

Project Kick-Off: Outline of Your Project's Security Activities

When starting a brand-new project, you usually begin with a kick-off meeting. At this meeting, you should have all the key stakeholders for the project present, and the purpose of the meeting is to set expectations for the rest of the project. A security representative from your organization should be present at this meeting. This person will explain where you can find any security requirements that would apply to this project, exactly which security activities (such as a penetration test, a threat modeling session, etc.) will need to happen during the project, and what the acceptance criteria are for the project being complete and ready for deployment to production (from a security perspective). If there are any policies that apply to this project, any specific technical security

directions or documents you should read, or anything that you need to know to make your project a success from a security perspective, this is the first chance to define and communicate this information to you and the rest of your team.

Ideally, also at this meeting, the security representative will start to build trust and new relationships with the people on this project, assuming they do not already know everyone involved. They are going to be working with this project team for months, possibly years, and you all want to set off on the right foot. Making expectations perfectly clear and planning ahead so that there's no panicking and no unnecessarily stressful parts of your project will make for not only a better end product but also a happier project team.

BOB SHIFTS LEFT

Bob remembers his 20s and the tons of overtime he always had to do. He remembers all the programmers working long hours, sometimes even overnight, to fix last-minute bugs before the end of projects. As Bob eventually decided not to be a software developer and to become a project manager instead, he vowed to himself that he would do whatever he could to stop that from happening to his teams. If he knew there had to be a penetration test at the end, that meant he needed to schedule time for the developers to fix the bugs he knew they would find. He also knew that if they "shifted security left," doing security activities earlier in the project, they would have *way* less work at the end. Bob became extremely conscientious about giving the software developers tools so they could check for security problems themselves and fix them earlier. He also made sure that he scheduled extra time for them to fix whatever was found during a threat modeling session, a secure code review, or any other security activity. Throughout Bob's project management career, he has been praised on multiple occasions for having his end products be extremely secure and for his projects usually being on time. Bob tried to tell everyone that all they needed to do was "shift security left" and schedule time for the developers to fix the stuff they found, rather than assuming they would just work evenings and weekends, but most people didn't listen. Bob still doesn't understand why more people don't manage projects the way he does. Maybe he needs to learn to explain his thoughts a little better if he is going to try to change people's minds.

Project Scheduling and Planning

We need to schedule security activities into the official project schedule. We need to plan enough time to mitigate any security bugs or flaws that we find in our product. It is not enough to just schedule a security activity to happen; we need enough time to fix the things that we will most certainly find *during* the security activity.

I don't know about you, but I prefer to keep my evenings and weekends for doing fun things, rather than working. It turns out, if you ask a whole bunch of people, including software developers, pretty much everyone agrees that they like to have leisure time and do not prefer to work 60–80+ hours per week. When we don't schedule enough time in our projects for security activities, we end up with poor results. When I say "poor results," I mean not fixing many of the security problems that we are aware exist, everyone having to work a lot of overtime, the project being late, the project costing significantly more than it should have, etc. There are many, many ways that a project can go wrong when we don't plan properly in advance.

You might be thinking, "But I'm just a programmer, not a project scheduler or project manager. I don't have control over this." You may not control the project schedule, but you can certainly point out that things are missing and ask questions. This is a chance for you to try to fix things before they have gone too far off track. When you look at the project schedule, if you see that there's something missing, ask about it. "I thought we were going to have a pentest as part of this project, but I don't see it on the timeline; why? Also, we need to add time to fix the bugs we will find, too." You might save yourself a lot of late nights at work.

DUELING PROJECT MANAGERS

Alice once worked on a project that was extremely high profile. It was so high profile that there were *two* project managers, and they both kept barging into everyone's cubicles all the time, without a meeting, demanding updates. Then they had a security incident, and the project managers became even more frantic. Several times per day, they would be in the cubicles of the incident responders (IR team), wanting to know when the security incident would be cleaned up. Eventually, the incident manager got upset with them and yelled that she spent way too much time giving updates and not enough time investigating and to "get the heck out of my office." Alice thought it was pretty hilarious at the time and quite obvious that interrupting the investigators over and over again wasn't adding any value. But now that Alice is older, she looks back on that and thinks, "My gosh, no wonder there's so much burnout in this industry."

Security Requirements

When you're a member of a technical project, you need to have technical requirements so you know *exactly* what to build. Ideally, with any technical project, there should be technical *security* requirements as well. By this, I mean specifics on what security settings, configurations, defenses, precautions, etc., that you need to make to conform to any security policies, standards, or guidelines your

organization has and to ensure that the product holds the security posture that your organization demands. The more specific these requirements can be, the better!

Project requirements must be documented and generally also signed off on. If you don't properly document the requirements and get approval (signoff), later, your decisions and end product might be in dispute. Ensure that every stakeholder reviews the requirements, gives feedback, and then signs off before you start the next phase of the project.

What follows are multiple potential security requirements for various types of projects. The lists are far from complete; they are just a starting point. Please feel free to go back to *Alice and Bob Learn Application Security*, Chapter 2, to see a list of web application security requirements and a deeper dive into this topic.

Web application requirements:

- All traffic to and from the application must be encrypted (web traffic sent over HTTPS), and all encryption must follow the encryption policy (using a secure protocol, such as Transport Layer Security [TLS]), including algorithms and key lengths.
- Data must be encrypted at rest.
- You must use the newest supported, stable version of your programming language and framework of choice.
- Follow your organization's secure coding guideline or standard.
- All code must be scanned with your static analysis tool before going to prod and before each QA cycle, and all reachable, medium, and above issues must be resolved. All non-reachable issues that are high and above must be resolved.
- Create custom error messages that follow your error handling, logging, and monitoring policy, available [*here*]. (Note that you always want to direct people to the other documents you are referring to with a link, if possible). Add more requirements for your organization, this list is not complete.

Web API requirements: everything in the web application security requirements, plus the following:

- All web APIs can only be accessed via your API gateway. How to configure it is documented via document [X], which details rate limiting, resource quotas, input validation, authentication, and authorization.
- All secrets, including your API keys, must be stored in a secret management tool and accessed programmatically. Secrets may not be stored in your code, the secret scanning tool must be run on all code, and the code must pass before it goes to production.

- Ensure that your error messages do not give very much information away, especially system information. Remember: only attackers or testers should see these messages.
- All APIs must use your service mesh. How to configure it is documented via document [X].
- Dynamic analysis of the API will be done via an API-specific testing tool, by the security team, before going to prod. You must remediate any security issues found that are of medium severity or above.

Add more requirements for your organization, this list is not complete.

Database requirements:

- The database must be set to encrypt all data at rest.
- All data must be classified for sensitivity and then labeled.
- Use strong, unique, random passwords for all database accounts.
- Enable MFA for all database accounts that have access above read-only.
- The database server must be hardened, with hardening configurations verified before going to production and then annually as a minimum.
- All user accounts for applications must be either read-only or CRUD (create, read, update, delete). Database owner (DBO) is not allowed without a written exemption from the security team.
- All calls to the database from APIs, web apps, or other systems must be done via parameterized queries. Commands cannot be sent to the database for direct execution unless it is done by the database administrator, via proper channels—never via an API, serverless, or web application.
- The database server must be patched and updated regularly, according to internal patching policy.

Chapter Exercises

You can find the answers to these questions in Appendix B, “Answer Keys.”

1. Which security activity would you like to implement in the requirements phase?
2. Name one security requirement you would like to have for a web application, API, or database. Or name one for each!
3. Why do we need security requirements for projects? Why isn’t it just obvious? Why do we need to document it?

4. In your own words, why do you feel a project kick-off meeting is important?
5. What would you like to know from the security team at a project kick-off meeting? What would you ask them if you could?
6. In your own words, why should we add security activities to project schedules?

Just as every project needs to have a list of requirements, we also need to plan out what we are going to build, and that means *designing* the software. In the Waterfall methodology, there's a formal phase for design, whereas in Agile or DevOps, we often design one feature at a time after the main part of the system is built. Also, for Agile and DevOps, we involve the end users a lot more when it comes to the design activities, constantly asking for feedback to ensure that we are on the right track. DevOps is also more of a cultural change than just a methodology. It requires new tooling and is quite a shift, but I definitely think it's worth it.

In this chapter, we are going to talk about security activities that you can perform while you are designing software. Most of these also apply to hardware and other types of physical systems you may be designing, but we will focus on software here. You do not need to do every one of these activities for every single system you build, but the more of them you do, the more secure your end product will be. You will need to decide with your security team how much security is enough for the importance of the system you are building, your organization's risk tolerance, etc. This is not something that a software developer will decide on their own; it needs to be someone who has the authority to do so and the experience and knowledge to make a good decision on behalf of your organization. Although as a software developer, you may not get to decide which activities do or do not happen at your organization, you will play a very

important role by (hopefully) being a willing and enthusiastic participant and a knowledgeable subject matter expert (SME).

Threat Modeling

Threat modeling is a (mostly) manual process that enables you to identify threats to a system. It's a group activity, ideally involving one or more technical representatives from the project team, a security professional, and someone representing the user (often called a *product owner*).

Threat modeling is a family of activities for improving security by identifying threats, and then defining countermeasures to prevent, or mitigate the effects of threats to the system. A threat is a potential or actual undesirable event that may be malicious (such as DoS attack) or incidental (failure of a Storage Device). Threat modeling is a planned activity for identifying and assessing application threats and vulnerabilities.²⁸⁰

— OWASP

Threat modeling can be incredibly intensive, with several meetings and extensive paperwork. It can also be just one session with one document as a result. Usually, organizations will start with a simplified process and add more to it as time goes on. It is more important that the process you use meets your needs and is something your organization is able to maintain, rather than it being perfectly aligned with a specific methodology. A good process that you follow every single time is much better than a perfect process that rarely happens.

The four key questions a threat modeling session wants to cover, according to Adam Shostack's Four Question Frame for Threat Modeling,²⁸¹ are

- What are we working on?
- What can go wrong?
- What are we going to do about it?
- Did we do a good job?

The first three questions are meant to uncover threats and brainstorm mitigations. The fourth question is to make the group think about whether or not the decisions made during the session are appropriate. Stepping back and taking extra time to evaluate the mitigations as a whole often results in changes to the overall plan for the better.

Although there are several different threat modeling methodologies that formally lay out exactly what to do, starting with these four questions is the easiest way to get going right away. Well-known threat modeling methodologies include STRIDE, PASTA, Attack Trees, DREAM, Trike, VAST, and so on.

If you are invited to a threat modeling session, I suggest you bring along a data flow diagram and any design documentation you have. Also bring your imagination! One of my favorite questions to ask during a threat modeling session is, “If you were going to hack your app, how would you do it?” Think about this before you arrive. How would you break into your own application? What’s the worst thing that can happen? Is there anything you are secretly worried about? How would you try to abuse your system? Are there undocumented workarounds? If you were an evil version of yourself, what’s the worst you would do? Bringing these ideas to the session can be very helpful, especially if you find a real threat!

During the session, the security person leading it will likely ask you several questions about how things work, where and how things connect, whether security controls are in the places they should be, etc. Try not to be too alarmed when they come up with rather frightening attack scenarios; it’s their job to think like a malicious actor. I remember being in my first threat modeling session and thinking, “How could anyone ever do that??? That’s *terrible!*” Although it took me a few years before I got good at it, I have found threat modeling to be a very enjoyable and enlightening activity.

The most important part of threat modeling is taking action afterward on what you have uncovered. There’s very little, if any, value in identifying several terrifying threats and then doing absolutely nothing about them. You need to eliminate (prevent) or mitigate (reduce the damage of) them. You may need to make design changes to add extra layers of defense, and you will definitely want to tell your testers about the threats so they can try to test for them. When you review your code, you can also look for threats for yourself or your colleagues.

Follow-up steps to a threat model include the following:

- Document the list of agreed-on mitigations, and then get everyone to sign that they agree. Save that doc with the rest of your documentation for the project or application.
- Set a schedule or deadline for when each mitigation will be implemented. Set check-in dates to follow up on each item to ensure that they are fixed and not just shoved into the backlog.
- Assign someone to review the code for the threat models you decided not to mitigate, to see if they are possible. If they are possible, you may change your mind about whether they need to be fixed or not.
- If you have a penetration tester, ask them to test for each of your threat models.
- If you make major changes to your design, whether because a lot of time has passed or because of change requests from the product owners, do another threat modeling session to find out whether additional threats need to be accounted for.

Ask your security team about threat modeling and see if you can attend the next session they hold. I promise it will be a very interesting meeting!

Threat modeling was also covered in Chapter 2 of this book, as well as in *Alice and Bob Learn Application Security*, the first book in this series.

Secure Design Patterns and Concepts

When designing systems, we can choose to follow design patterns and concepts we know are more secure, as mentioned already in Chapter 4, under the heading “Secure Design.” As this part of the book is about the secure system development life cycle, in this section, I would like to ask you to try to apply what you previously learned when you are designing systems. This means when drawing out a design, writing up a design document, or attending design meetings, suggest the most secure pattern or concept that you think might fit the situation. This also means pushing back or resisting if someone pushes you not to add an extra layer of security or not apply a concept you feel is needed. You may need to emphasize the value of security and the risk to your organization more definitively. Although you may not be the one making the final decision, offering suggestions for how your team can do better and pushing back if you feel they are cutting corners can go a very long way toward your team making better choices.

Another thing you can do is talk to the security team. Invite them to your design meeting or to review your diagrams. Ask for their input. Also, talk to your team lead or project leader and ask how you can ensure that there is a chance to apply one or more of these patterns or concepts.

Often, when a security activity is not performed, it’s not because the team actively decided not to do it (due to limits in time, budget, or resources) but rather because no one thought of it. Just suggesting a pattern or concept at the right time can go a long way as long as it makes sense for your project.

NOTE The specifics of this topic were covered in detail in Chapter 4 of this book, as well as in *Alice and Bob Learn Application Security*. In this section, we will talk about how to make sure you have the chance to apply what you have learned in a work situation.

NOTE What do you do if your organization does not have a security team? This leaves you, the software developer, at quite the advantage. If you work in a very small business, you can likely go talk to the head of IT and tell them about your security concerns, plans, or ideas. If there is no head of security or security team at all, that responsibility falls to the CIO (chief information officer). If your company is so small there is

no head of IT, then it falls to the CEO. The CEO accepts all the risks for the organization, no matter how large or small, and they need to be aware of what's going on. Although they might not be pleased to hear about a huge risk you found, I assure you that they would rather know than not know. Or even worse, find out via a press release or a ransom letter . . .

Architecture Whiteboarding

The purpose of an architecture review is to have multiple people look at the architecture you have planned for your project and help you improve it. Whiteboarding is the act of drawing the design on a whiteboard so that others can visually look at what you have designed. A whiteboard makes it very easy to fix any issues because you can just use your finger to erase the offending item and redraw it to match your new and improved idea.

Ideally a whiteboarding session is a conversation, with drawing. And you do not need to be an artist (I assure you, I am not an illustrator). They say “A picture is worth a thousand words” because it’s much easier for most people to understand what you mean when they can *literally* see what you mean.

Often a whiteboarding event is informal, with some brainstorming and problem-solving happening. Once you all agree that the architecture you’ve drawn is good, you will want to snap a photo of the final drawing with your phone; then you can begin writing up a formal architecture document.

Once your architecture document is complete, you will likely have some form of architecture review (this might mean a meeting or submitting your document and waiting for feedback), which *will* be formal. You will need to defend your decisions and answer questions. That said, if you already got input from several teams, you will be in a great place to get approval on your first attempt.

Examining Data Flows

A data flow diagram (DFD) maps out the flow of information for any process or system.²⁸² You will definitely want one for any piece of software you are going to create; not only are they handy, but they also help you plan your design *and* help you show others how the system works.

From a security perspective, the places where different parts of a system integrate are likely to have potential security problems. On top of this, you know that the security of your data (its confidentiality, integrity, and availability [CIA]) is of the utmost importance. Combining both of these pieces of information leads us to want to examine where and how our data flows throughout our systems.

Take your DFD and examine it, and look for potential security problems. If you have already done a threat model, you have likely already searched your DFD for threats; but if not, do it now as part of this process. Ideally, invite a security person to look at it with you. For each place that your data is moving from one place to another, you want to ask yourself some questions. Is the data in transit *encrypted*? Is there both authentication *and* authorization before you send the data? How do you know *whom* you are sending it to? Is the data *validated* once you get it, to ensure that you have received something good and safe to use? Where is the data stored? Is it encrypted at rest? Is it properly classified and labeled? Every place in the diagram that the data moves or changes, you want to ask yourself as many questions as possible to assure the CIA of that data. If something seems to be missing, add it to your design or architecture for your project.

Security User Stories

User stories first came into existence in 1989²⁸³ as part of Extreme Programming and Agile frameworks, which means they have been around a very long time and our industry should know them well. A *user story* is an informal document that briefly explains a software system.²⁸³ You write it from the end user's perspective, making requirements and expectations for every feature and functionality of the system as clear as possible. There is usually one story per feature, and it should explain the value of the feature as well as what it does.

A *security* user story tells the story of each *security* feature of the product.

As you might have expected, OWASP has quite a few amazing examples of helpful user stories, available as part of its User Security Stories page: <https://github.com/OWASP/user-security-stories/blob/master/user-security-stories.md>.²⁸⁴

Some examples of user stories could be the following:

- As a security-savvy software user (SSSU), I want to ensure that my data is safe when I share it with this product. Acceptance criteria: Set data to be encrypted in transit and at rest. Label and classify your data. Use parameterized queries to interact with the database. Use static analysis to validate your implementation.
- As an SSSU, I want to be safe using this application and not have XSS attacks happen. Acceptance criteria: Use JavaScript framework known for performing output encoding, test using a DAST and SAST scanning, avoid inline HTML/JavaScript, review code for XSS and/or improper encoding.

- As an SSSU, I want to ensure that no one can use my accounts or take over my accounts. Acceptance criteria: Threat model or perform design review, use a preexisting AuthN and AuthZ system, review code to ensure that this security control is present where it should be, and test with a pen test.
- As an SSSU, I want to ensure that the password I use for this system is kept safe. Acceptance criteria: Review the code for this feature, ensure that it is part of your design and project requirements that you will salt and hash stored passwords, and use a SAST to ensure that you choose strong cryptographic functions.

For every security requirement and security control that is part of your design, you should create a security story and acceptance criteria. These stories should be kept with all the other user stories and followed up on in the same way.

Chapter Exercises

1. Create three of your own security user stories. The first one could be about avoiding saving your secrets in your code. Come up with two more on your own. Don't forget your acceptance criteria!
2. Get a data flow diagram from work, or search for a simple one using Google image search, and then take a big, long look at it for security issues. Ask yourself, "Where would I want encryption? Where would I want to do authentication or authorization?" Circle things, add notes, and make a list of what *you* would want to add if it were your system.
3. Take that same data flow diagram and go to <https://shostack.org/games/elevation-of-privilege>. This is a free site where you can play the Elevation of Privilege (EoP) Threat Modeling Card Game, or purchase it. Try to play the online game using the image you found (you can switch images if the one you chose turns out to be too difficult). If possible, invite friends along for this exercise; it will be more fun that way.
4. Go back to Google Image search, and look for an architecture diagram of a web app with an API. Perhaps search for this: "web app architecture diagram with API." You will find several images; pick whichever one you want, but ideally one that comes with a description of what the system is supposed to do. Imagine yourself attempting to threat model this system: where would you start? Try to find two or three threats to this system. Ask yourself Adam Shostack's four questions; it will help! Then think about how you would mitigate those threats. Write it all down; don't just

think about it in your head, as writing it out will help you learn and remember.

5. Draw a pretend architecture for the following system, and then attempt to whiteboard it: a web app that allows parents to plan play dates for their kids. They can add their children and who their friends are, along with the friends' parents' phone numbers and weekly schedules, so the system can make play dates for children whose schedules match. Each parent would make an account and then "connect" with other parents of children their kids like. What risks could there be to this system? What data is sensitive? What if there are kids who don't like each other? Or parents who don't have a car/transport? You can go pretty far with this example if you want to, but you can also draw up a different system instead if this scenario doesn't fit right for you. The idea is you want to draw up a system, get used to drawing an architecture, and then finding problems with it and fixing them.

While you are writing code, you can do several things to ensure that the code you write is more secure, and that is what this chapter is about. Even if you have no security team to help you and no money for fancy tooling or training, you still have more power than you may realize. You have power over the way you write your code and some of the tools you use as you write your code, and even if testing is not part of your job description, you can run tests against your code. I'm sure you've heard the phrase "Knowledge is power"; well, in this chapter, I am going to arm you with knowledge so that you can take matters into your own hands.

Training

When I was a developer, I rarely had access to training other than my computer science diploma, which I paid for myself by working two jobs throughout most of college. When I wanted to switch from being a developer to working in security, I started speaking at conferences so I could get in free. With the exchange rate of the Canadian versus American dollar being rather uneven for most of my adult life, and most of the training available in English being from the United States, that meant I could afford almost no training at all. I remember doing the math and figuring out that one SANS Institute course plus one certification attempt would cost approximately 20 percent of my after-tax take-home pay

for the year. That meant no going out to dinner. No movies. No concerts. For an entire year. That felt like too big a sacrifice to me.

I had a whole strategy for how to learn enough and get enough experience to switch careers, besides just speaking at conferences. Back when I was beginning my career in AppSec, the OWASP Foundation used to save a free spot at every conference training for one project leader and one chapter leader; I was both, so I got free training that way as well. I also soaked up every bit of free learning on the internet that I could find, from listening to podcasts to reading blogs to following courses on various online academies (which were of greatly varying levels of quality). I took out every “cyber” book from the library and read it, as well as bought several textbooks, doing all the exercises inside using my own home lab. On top of all that, I had multiple professional mentors throughout that career change, and each taught me different things. It was not exactly an efficient way to switch careers (it took years), and it was quite difficult at times, but it worked, and it helped make me the security professional I am today.

When I was a software developer, I made every security mistake that I now tell you not to do in this book. In my computer science program in college, they taught us to save the secrets into our code, not to validate or sanitize our inputs, and to design our systems with implicit trust. Back then, we just didn’t know any better. Today, most degrees, diplomas, and bootcamps barely teach you any security at all, graduating new developers almost every day with little to no secure coding knowledge.

TIP If you are a student at a college or university, ask your professor if they will cover application security or secure coding as part of your program. If they say no, ask why not. Ask them to add it to the program. Tell them it’s important to you. If enough of us ask, it might change for the better, faster.

I’m telling you all this not so you will have feelings about past-me working hard to learn, but so you will understand *just how much* work it is to try to do this on your own (outside of work) and that most software developers just aren’t going to do it. They aren’t going to pay thousands of dollars from their own pocket, study every night, or use their vacation time to attend a conference. They have lives outside of work, and to be frank, I think that’s normal and the way it should be. Work-life balance is important, and I certainly hope you do not take lessons on that topic from me, as it is something I struggle with fiercely.

If you need the software developers where you work to create and maintain secure applications, you have to teach them how. You cannot assume that they will show up already knowing how or caring (no matter their age or where they learned to program). If you plan for them to learn “on their own time and dime,” you are bound to be very, very disappointed. Your organization needs to support them in their learning, and that means training.

There is good news, though. Training can take many forms, there are significantly more options now than ever before, and the price has come down dramatically since I joined AppSec a decade ago. What follows are a few strategies that you and your organization can take to learn more about secure coding and how to create secure software.

NOTE I give secure coding training. You can hire me to teach your team everything in this book and more. Email for more info: tanya@shehackspurple.ca.

Organizations

When buying in a group, you have more buying power. This means you can bring a teacher in to teach you, and most of the time, that costs less per person than sending your employees for training at a private institution or paying for training at a conference. The teacher can also customize it to your organization's needs, and everything your team says during the engagement will be covered under an NDA. If you work for the government, you can partner with other departments or partner organizations to get a group discount if you're organized about it.

Some organizations purchase licenses for cybersecurity training platforms that have either interactive lessons or on-demand lessons. Although I personally really like these because I can take my time and learn a bit each day, not every person is so diligent. That said, I have bought many online courses in which I only did the first couple of lessons and then stopped, despite my best intentions. If your organization takes this path, it is very important to have a plan for how to get your employees to use the system, as it is common for most of them to use it only once, with few completing all the lessons unless either it is forced on them or you incentivized them to do so. On top of that, ensure that they have time to complete the lessons; expecting them to do it on the weekend will often lead to disappointment on your part.

A tactic I have seen work at some organizations is to start a developer education or security champions program, both of which I have spoken and written about quite a bit. A developer education program means the internal staff plans a curriculum for all the developers, and they can create the content themselves or bring someone in to do it. They often do smaller teachings, regularly, to ensure that they keep the engagement high. A security champions program is similar in the way it's taught, but the focus is on one representative from every developer team, who becomes their "champion" to secure their applications; generally you teach them a lot more. I realize I am not doing either of these topics justice with this short description, but you can learn more easily by either taking my "Application Security Foundations Level 2" course for free from Semgrep Academy or doing an internet search for "SheHacksPurple" and either "developer education" or "security champions"; there will be many, many pieces of content to help you create a great plan for either program.

Individuals

I remember the first time I had a dynamic application security testing tool (DAST) scan run on one of my apps when I was a developer. I searched for “cross-site scripting” so I could learn how to fix what it had found, and there were only three results on the entire internet at the time! My how things have changed—for the better!

Free training is now more abundant than ever before. You can learn secure coding, web app hacking, and much more with just some internet searches and time. I strongly suggest that you focus on secure coding, secure design, and secure code review if you are a software developer. If you are looking to become an application security professional someday, I suggest learning how to make an application security program and then diving in deeper to learn more about your favorite parts of an AppSec program (such as threat modeling or penetration testing). Learn what the options are, and *then* make the rest of your learning path.

To ensure that you keep learning, rather than stopping part way through, you need a strategy that works for you. This might mean setting aside some time every day or a few times per week for learning, getting an accountability partner, and/or finding learning formats that work best for you (I prefer audio-books over physical books, for example). If you have a copy, I suggest going back to Chapter 10 of *Alice and Bob Learn Application Security* to help you create your continuous learning plan.

NOTE You can continue to learn from me for free! I have formal lessons over at Semgrep Academy, <https://academy.semgrep.dev>, as well as less formal videos on my YouTube channel, <https://youtube.com/shehackspurple>. You can ensure that you never miss a single piece of my content, lessons, or events that I will be part of by joining my newsletter, <https://newsletter.shehackspurple.ca.-D>

Code Review

Secure code review means analyzing an application’s source code and looking for security vulnerabilities. It can be manual or automated, but a combination of the two will give you the best results.²⁸⁵ Code review is sometimes called *white box* or *clear box* testing, but more commonly, it’s known as *static analysis*. You only need read-only access to the code to do a manual code review or run a static application security testing (SAST) tool.

If you are going to perform a manual code review, you should ensure that all the security controls the design calls for are indeed where they should be and implemented properly. Also, look for your threat models in the code, business logic issues (flaws), and implementation issues (bugs).

If you are going to perform an automated code review, you will want to use a SAST tool, also called a static analysis tool. According to me, there are two types of static analysis tools: first-generation and second-generation. First-generation tools perform a combination of symbolic execution and flow analysis, which is slower and produces more false positives but is extremely thorough. Second-generation tools are much faster and produce fewer false positives, but they miss things. Second-generation tools perform a mixture of antipattern matching and flow analysis. With most organizations having hundreds of bugs in their backlog and prioritizing fast feature release over security, second-generation tools have quickly been taking our industry by storm. To find out if a SAST tool is first- or second-generation, ask if it performs symbolic execution or antipattern matching.

First- and Second-Generation Static Analysis Tools

The place where you work may or may not provide a static analysis tool for you to use. If it does, please use the tool! Ideally, it will have an IDE plugin that can help you as you code; but if not, at some point you should see a report with bugs in it from the security team. If your workplace does not provide a static analysis tool, there are multiple free tools you can use (such as Semgrep Community, Brakeman, Bandit, Horusec, and so on—there are *many*) that can help you analyze your code for vulnerabilities. You should use at least one to analyze your code and then fix anything that seems disconcerting. If you aren't sure, ask the security team at work. If you don't have a security team, ask in the OWASP Slack channel; there are a lot of friendly AppSec folks there.

NOTE First-generation SAST tools still have their place in our industry, even with companies all rushing toward DevOps as they “move fast and break things.” Sometimes you need your application to be as close to perfect as humanly possible, such as if you are doing counterterrorism work, defending your nation's democracy, or waging war. There are times when it makes sense to spend \$1 million securing just one application, such as when there are human lives at stake or catastrophic risks. Sometimes you will want to use *both* kinds of tools because you want to be extra thorough. Do not dismiss these tools just because they are not well-adapted to the current breakneck speed of software development; they have their time and place and still offer value.

FIRST-GENERATION SAST	SECOND-GENERATION SAST
Fortify	Semgrep Code
Veracode	Snyk Code
Checkmarx	QuietAI

TIP Although there are many cool tools out there that claim to be “free,” some of them have limits built in. They aren’t truly free; they are free trials, which is great for trying a tool before you buy but not great for ongoing use. Check the license before you dive in too deep to ensure that (1) you are compliant and (2) you don’t get cut off just as it gets good.

NOTE It is my experience that the most obvious difference between paid and free SAST tools is the quality of the results you receive and how well they work with the rest of your tools. Free tools rarely offer an API or integrations for your CI/CD, a code repo, bug tracker, and other dev tools. Free tools usually also have a *lot* of false positives. If you work in a place where you have no budget at all but lots of time, you can do well with free SAST offerings (OWASP has a page that lists all of them here: https://owasp.org/www-community/Source_Code_Analysis_Tools). But if you have lots of work and little free time, you will receive far better results faster with a paid SAST tool. The same goes for secret scanning tools; you pay to receive fewer false positives and false negatives. That said, the same is not true for DAST tools, where my two favorite ones (Burp Suite Pro and Zap) are almost free and free, respectively.

TIP A code quality tool or linter is not the same as a SAST tool. Ideally, you would use *both* a quality tool (linter) *and* a SAST tool; but if you can only pick one, I urge you to pick the security tool (SAST).

Secure Guardrails

Some static analysis tools will allow you to build “secure guardrails,” custom rules that help ensure that you and your team make more secure coding choices. You can create guardrails that help enforce your coding standards, such as using one specific validation function over another, using preferred naming conventions, or using CamelCase rather than snake_case. You can also use these tools to find and prevent serious security bugs or flaws to ensure that they never happen again. Guardrails can be implemented using a SAST tool via an IDE plugin, when you check your code into your version control, when you run your CI/CD, or any other time you think it will get you and your team back on track to write better code. The idea of a secure guardrail is the same as a physical guardrail on a road: to inform you that you are on the wrong track and help steer you back to the paved road where you should be. You are still able to continue and go “off-roading,” choosing not to use the secure defaults or standards of your org, but you are doing so consciously, and there is a reason for it. Creating and using secure guardrails helps avoid many common mistakes, saves time and money, and is generally considered a proactive approach to security. If you decide you want to try it out, work with your security team for the best results.

IDE Plugins and Other Guidance

Your IDE is an extremely powerful and amazing tool. When I first started in college, I remember we used Borland, and compared to the current tools that exist, such as VS Code, it was *terrible*. There are many amazing plugins that can be part of your IDE and help you with many different things. There are static analysis plugins, linting tools, tools that help with your APIs, and tools that will proxy your web app for you; there are many great tools to choose from! It's wonderfully overwhelming.

I suggest using the following plugins (from whatever vendor makes sense to you):

- Static analysis: Ideally one that puts red squiggly lines under questionable code.
- Secret scanning: To tell you if you made a mistake by putting a secret in your code.
- Software composition analysis: To tell you if your dependencies are known to be insecure and which other versions might be a better choice.
- Linter: Increasing code quality is always a good thing.
- IaC scanner: Checks your infrastructure as code for bad configuration choices.

Run them at least once before you publish your code, and fix the issues you find. I realize you might not want to fix every single thing they find, but at least fix some of the things.

TIP Not all plugins are created equal! Please verify that whatever company makes the plugin is trustworthy before you download or install it. It's unlikely that a well-known security vendor is going to intentionally release a malicious plugin, but a plugin built by an individual (unless they are very well known) or an independent, unknown, open-source project should probably be avoided (unless you test it thoroughly and your security team approves it).

TIP I used to recommend that people make unit tests for various types of security vulnerabilities, such as cross-site scripting (XSS), until I discovered that I could better scale this idea using SAST custom rules and then turn it into a secure guardrail to the same effect. You can create one custom rule per language, per vulnerability, and have it test your entire code base; whereas if you make unit tests, you have to make several tests per vulnerability, and each of those tests needs to be repeated for every single unit of code that vulnerability could apply to, which just doesn't scale. There are also community-written custom SAST rules available on the internet for popular tools and languages, written by people like you!

Verifying That Your Dependencies Are Safe (SCA)

As previously discussed in this book, all the code in your application represents risk, even the code you did not write. If you are going to use a framework, a NuGet package, a Ruby gem, or any other code that you did not write, ideally you want to ensure that it is safe to use. The easiest and most reliable way to do this is by using a software composition analysis (SCA) tool to determine which dependencies have known vulnerabilities in them and then decide which ones are worth upgrading off of or otherwise removing and which you shall keep.

How Do You Decide Which Dependencies Are Worth Updating or Changing?

According to a post by Jonathan Eunice on StackOverflow,²⁸⁶ you should generally upgrade dependencies when

1. It's required.
2. There's an advantage to do so.
3. Not doing so is disadvantageous.

When I was a junior software developer, I definitely agreed with this statement and generally stuck to #1. As I became more senior and had to dig my team out of extensive technical debt, I started to find ways to ensure that I was doing small upgrades all the time to avoid future pain (#3). That was before I struggled as an incident responder due to dependencies that were so out of date that we just could not respond properly. If it's going to take one or two years to upgrade off of a grossly insecure dependency that is currently being exploited in the wild, that's not an acceptable outcome from a security perspective. In summary, over the years, I have changed my mind and now feel that #3 applies almost constantly.

Currently, I try to encourage software developers to keep their dependencies as up to date as they can, focusing on ensuring that they are always on a supported version. On top of this, I run SCA tools against our code base and ask to have dependencies upgraded that have medium, high, and critical vulnerabilities that are reachable or exploitable. If a vulnerability is critical, I ask to have us upgrade off of it, even if it is not reachable, because it's essentially a time bomb inside the software.

If you are looking at upgrading a dependency and you realize it will be extremely difficult, time-consuming, or expensive to do so, talk to your security team. Explain the cost, and ask them to help you decide what to do. Sometimes, there are other solutions you may not have thought of.

OUT-OF-THE-BOX SECURITY

Alice remembers a project management contract she did with a large IT company that found two huge vulnerabilities in the company's dependencies. The apps they were found in were very old, and it was going to require a complete rearchitecting of both apps to upgrade off of the dangerous dependencies. One application served hundreds of customers, was hosted by them, and was open to the internet. The second application had only four customers, with plans to decommission it in the coming year. After much discussion, the team decided to rearchitect the first app; they put a Web App Firewall (WAF) in front of the second app at each customer site until they decommissioned that app as planned. Alice's team decided this reduced the risk to an acceptable level and cost less than rewriting an old app that was on its way out. The client was quite pleased with Alice's out-of-the-box thinking.

Finding and Managing Secrets

Secrets are confidential strings that are used for machines to authenticate to each other. A secret could be a password, an API key, a certificate, and so on. Any kind of long, randomly generated value that is kept private can be used as a secret. As I said previously, when I was in college I was told to save secrets directly into my code, but that is no longer appropriate. There have been many well-documented security incidents where malicious actors have been able to take secrets from source code and then use them against us to steal our sensitive data, to take over our systems, you name it. As a result, our industry moved to create systems that were specifically made to protect those secrets, and those systems are called *secret management tools*. It is no longer considered appropriate to store secrets directly in our source control; now we should access them programmatically, taking them from our secret management tool.

If the organization you work for does not have a company-wide secret management plan, and it is likely that the company has not yet searched its entire code repository for secrets, I suggest you start with that. There are many secret scanning tools available, some free and some paid. As mentioned previously, the paid ones will have significantly more accurate results, which will save you a lot of time, but they also cost money, so that's a choice for you and your budget to make.

Once you get your hands on a secret scanning tool, scan your entire code base. It will take a surprisingly short time before it starts finding secrets. For each secret, you will need to validate whether it is a real secret or not and whether it is live. Some things are not real secrets; for instance, a secret scanner might pick up the word "password," but it's actually a user password variable (which is not considered a secret), or it's just the word in a sentence. You will want to eliminate all the false positives from your results; then validate which secrets are live. A live secret means that it is currently in use, and thus you need to prioritize fixing those first.

SPILT SECRETS DON'T STAY SECRET VERY LONG

Bob heard from his OWASP friends that open-source repos on GitHub are scanned *every 30 seconds* to see if a secret has been leaked! “Holy smokes!” he thought. He made a special note to set up the precommit hook for their new secret scanner the very next day.

When you find a secret, you should rotate it, which means generating a new one. If you *do not* have a secret management tool at work, rotating the secrets will at least mean that if someone else has already stolen it, you have ruined their fun, and they will have to try to steal the secret again. It reduces the risk slightly, but not very much. This is your chance to really shine in front of the security team by showing off your list of secrets you’ve found and asking them to buy you a secret management tool so you can manage them more safely. Hopefully, they say yes.

If you *do* have a secret management tool, you should rotate every secret and then check them into the secret management tool. After that, you will need to change your code so it grabs the secret from the secret management tool instead of where it used to find the secret in your code. The first time you set this up, it takes a bit of effort; but once you’ve done it, it’s easy to just copy and paste.

Now that your code base is clean (no more secrets), you will want to ensure that you block everyone at your org from ever checking a secret into your code again. First, tell everyone what you have done and why secrets in code are bad. Then tell them how they should be handling secrets from now on (show them how to use the secret management tool). Then set up a precommit hook in your version control system that does a secret scan and blocks anything that it thinks might be a secret. Err on the side of caution: if it looks like a secret at all, block the check in. Tell everyone that you are going to block potential secrets at check-in so they understand what’s wrong if they see that message and they know what to do about it. If you do all of this, you are basically a security superhero!

By removing all the secrets from your code and accessing them programmatically from a secret management tool instead, you will greatly reduce the risk for your organization. This is something to put on your resume; I would be quite impressed if a software developer told me that they did this of their own accord. Make sure you tell your security team; they will be very pleased with you.

Dynamic Testing (DAST)

When I first learned to become a penetration tester, the very first tool I learned to use was a DAST called Burp Suite Pro. Burp is extremely well-known, wildly popular, and a very high-quality tool. It can also break a lot of stuff if you don’t

know what you're doing, which, unfortunately, a couple of my clients and I found out the hard way as part of my junior pentester journey. Because dynamic scanners include fuzzers, a feature that most of them have, sometimes they're able to cause damage to applications and databases by accident. If you do not have someone who is extremely experienced in using them, it is not a good idea to let just anyone use such tools against a production system. Generally, it is the security team who runs the dynamic scanners rather than software developers, but I worked at one place that created a sandbox with test data for the developers to use dynamic scanners as much as they wanted. They found *a lot* of bugs.

Dynamic scanners are pieces of software that interact with your running web application or API so that they can try to find vulnerabilities. Your app has to be running (live) for the scanner to work. It does not need access to the source code and is sometimes called *black box testing*. DAST scanners tend to work better on web apps than APIs unless you buy one that is specifically made to work on APIs only or you are using more advanced features than just doing an active scan.

DAST scanners tend to have several different features, but the main differentiators are whether they only do automated scanning or if you can also use them for manual testing. The ones that allow for manual testing as well are often used by penetration testers and tend to cost less than the fully automated ones (no, I do not know why the ones with more functionality cost less). The ones used for manual testing are also the ones that can cause the most damage if you are not careful. Examples of DASTs that allow for manual testing include my two favorites previously mentioned, Zap and Burp.

If you decide to learn how to do dynamic testing, it is extremely important that you have permission to test whatever you are testing, in writing, from someone who has the authority to give you permission. It is important that if you do have permission to test a system, you also write out a scoping document that states exactly what you are and are not allowed to test. You could end up in jail, depending on the country you live in, if you attack a web application. If you do not have permission to test a system and you point a dynamic scanner at it, you are, under the law in most countries, *attacking* that system. *Please do not break the law.*

If you decide that you want to start using a dynamic scanner and learn about dynamic testing, I suggest starting with Zap (Zed Attack Proxy), which is free, or Burp Suite Pro (which is priced very competitively compared to the rest of the DAST tools on the market, approximately 2–3 percent of the cost of an enterprise DAST). Once you decide which tool you want to use, you can either fool around with it in your own home lab, as I did back in the day when there were not very many teachers who shared information on these topics, or you can find someone amazing to teach you and learn a lot faster. I suggest checking out content by a friend of mine named Rana Khalil, who shares a lot of free lessons on this topic and is particularly great at explaining things. She will help teach you how to create a home lab so that you can test safely on your own computer, meaning the only permission you would need is your own.

Once you have learned how to use a dynamic scanner and you feel pretty confident about it, you should ask the security team if you would be allowed to test one of your own applications in a development or test environment. Explain how much you've learned and your intention to find bugs in your app so that you can fix them, and hopefully their eyes will light up with joy. Even if their eyes don't light up, hopefully, they'll let you apply what you have learned. There's a good chance they will be hesitant about letting you use such a powerful tool at work, so make sure you proceed with extreme caution if you do get permission.

TIP Never run a dynamic scanner against an app that you do not have permission to test. It might be illegal, and it's definitely rude!

Chapter Exercises

You can find the answers to these questions in Appendix B, "Answer Keys."

1. Is it a good idea to use a dynamic scanner (DAST) against any app on the internet? If so, why? If not, why not?
2. Are you allowed to use a SAST (static application security testing) scanner against any open-source code that you find on the internet? If so, why? If not, why not?
3. In your own words, why is it important to try to keep your dependencies up to date?
4. Which IDE plugins do you currently have? Which ones do you want to get?
5. What is a secure guardrail? Explain in your own words, and if possible, give an example.
6. In your own words, what is the purpose of code review? What is the benefit?
7. Take a few minutes to write down one or more topics that you are interested in learning and how you plan to try to learn them. Talk about it with your boss or team lead at work. If you are not currently employed or are employed at a place where you cannot learn these skills, tell a friend what you are trying to learn so that they can hold you accountable and help keep you motivated.
8. Which of the tools mentioned in this chapter do you have at work? SAST, SCA, Secret scanners, linters, DAST, etc. Find out if you don't know, and then ask the security team if you can use some of them. Put your experience on your resume; it's valuable!

Depending on your experience, you may have been looking forward to this chapter about security testing for a very long time, or you may have been dreading it. I personally like to think of penetration testers the same way I think of QA testers: they are finding as many bugs as they can so that I can fix them *before* the customer sees them. Although their reports full of bugs add to my workload, they also help me avoid any future embarrassment or harm if someone who isn't as ethical as we are finds our bugs instead of a trusted person who is on our payroll.

Throughout this chapter, we will cover the following topics:

- Test coverage and timing
- Manual testing
- Automated testing
- Fuzzing
- Interactive application security testing (IAST)
- Bug bounty programs
- Test results

By learning about these topics, you will be more prepared for when security testing happens as part of your projects. You will know what to expect, how to prepare, and potential outcomes.

We perform security testing for several reasons, and you might be surprised by the value it provides. When we find vulnerabilities earlier in the system development life cycle (SDLC), that means we can fix them earlier, and that not only saves us time and money but also often allows us to create better fixes that are of higher quality. We perform security testing to reduce organizational risk (which makes investors happy), for compliance purposes (satisfying auditors), to improve quality (security testing can find more than just vulnerabilities, such as memory leaks), and to protect our customers, employees, and citizens. By performing security testing and fixing what we have found, we help maintain our organization's good reputation and competitive advantage; customers are asking about the security of the products they purchase more now than ever before. No one wants to be a customer of "that company that had the scary data breach."

Performing security testing also affects the culture of the place where we work for the better. If we all know that our apps will be tested for security, we will work harder to make sure they are secure. We will also talk about security more often, and if we're lucky, that will make for a more positive and proactive attitude for everyone around us.

Performing security testing regularly and mitigating issues will also help us increase the reliability of the software we build, ensuring better uptime and resilience. Fewer bugs mean fewer crashes. Even if we decide not to fix every issue we find, documenting and being aware of them can help us be more prepared if something does go wrong, which protects everyone involved.

Regular security testing is a winning strategy for any organization that builds software.

Test Coverage and Timing

Test coverage in software testing refers to the evaluation and measurement of the degree to which a software application has been tested. It assesses whether all the critical components, features, and functionalities of the system have undergone sufficient testing to ensure their proper functioning and reliability. Test coverage is essential for identifying untested areas and minimizing the risk of defects in the software.

Testing coverage refers to what is being tested and what is not. It also refers to *how* it is being tested: in which ways. This can mean which of your apps have automated tools crawling over them on the regular and which parts of a system are included in the scope of a penetration test, but it can also refer to which angles the system is being tested from (static, dynamic, externally, internally, stress, etc.). It is important to ensure that we test from multiple angles, as well as the complete system, to ensure sufficient security and reliability.²⁸⁷ Thorough test coverage is critical for identifying untested areas and minimizing the risk of defects in the software.

If you are going to test a system, if possible, test the entire system, from the first access a client has to it (perhaps a login screen) all the way down to wherever your data is stored (usually a database but maybe a mainframe), along with any other systems it integrates with (APIs, other apps, third-party databases, etc.). You will want to be extremely explicit in the scoping document you use to define your test because if you break something that was not in the document, you might get into hot water. Obviously, you are not trying to cause damage to your organization, but it happens, and you want to ensure that you are safe from any legal repercussions.

Depth Versus Coverage

When determining coverage, something you will want to decide is whether you want to go deep (depth) into certain features, vulnerabilities, or parts of the system or if you prefer to have the entire system tested (coverage).²⁸⁸ There often isn't time for both if you have a consultant doing the test for you. I generally prefer to have wider coverage, rather than digging in on just one part, unless that part has critical risk (for instance, someone could die if the feature malfunctions or top-secret information may be leaked that could harm your nation).

Scanning Your Infrastructure

While you are deciding on the coverage of your test for your application, you should also determine whether the underlying infrastructure will be part of the test. Usually, the container, virtual machine, or Platform as a Service (PaaS) is scanned with a vulnerability assessment (VA) scanner to ensure that it is properly patched and configured, but a network penetration test is not as common. I always recommend scanning your infrastructure with a VA scanner, even if you are only responsible for the security of your applications, because if a malicious actor can hurt your organization via the platform your apps sit on, generally everyone will still blame you for the damage that ensues. This can mean ensuring that someone else on another team scans it or doing it yourself (if you have permission to do so). Please fix what you find, if at all possible.

Production or Lower-Level Environments

Although your test will be the most accurate if you test in production, there is risk in doing so as you may harm the production system or its data. However, if you are following good software development practices, your pre-prod environment should mirror production in every way except performance. This means a web app or API test should yield the same results (with the exclusion of performance, stress, or Denial of Service [DOS] tests). If your environments

do not match, I suggest creating a project to make them match. It is a worthy goal and offers significant value, not just for security.

When testing in production, it is crucial to create a backup immediately before the test of your database (always) and application (unless you do DevOps and are able to redeploy perfectly, quickly, and confidently). Ensure that you know how to roll back your data and app if damage occurs as part of the test.

Scoping

When writing up the scope of a security test (pen test or dynamic scanning), ensure that you explicitly state which pages, URLs, and IP addresses are included, as well as every type of user the system has (admin, guest, etc.). List all the APIs and other endpoints, as well as whether any other systems are included (such as Internet of Things [IoT] or mobile apps). You will also need to make a list of what is out of scope, such as specific endpoints, features, and certain types of attacks (denial of service [DOS] and account takeover are almost always out of scope) that are not to be a part of the test. State how long the engagement will last and how much it will cost if you are not having an employee do this work. Ensure that you get signoff on this document from whoever has the authority to approve this test before you begin testing.

TIP When arranging a penetration test, please do not “scope out” anything that you worry won’t pass the test that ruins the value the testing provides. This is quite common, especially for governmental departments. A rubber stamp that says you are secure is nowhere near as valuable as fully understanding the threats and risks to your software. “Plausible deniability” is no longer an acceptable reason for a data breach.

Timing

Although you should perform all sorts of security evaluations (static application security testing [SAST], software composition analysis [SCA], secret scanning, dynamic testing, etc.) throughout the entire SDLC, as well as after your app is released into production, a penetration test of a brand-new application should be done at the very end of the SDLC. It should be done as close to your release as possible to ensure that you are getting the most accurate version of the application while still leaving a realistic amount of time for the developers to fix whatever might be found.

If an application is already in production and you are performing a static (nondangerous) form of testing (SAST, SCA, secret, linting), you can test (and retest) the latest version of the code whenever you find it convenient.

If you are performing dynamic forms of testing (dynamic application security testing [DAST], penetration testing, red teaming, fuzzing, etc.), and the app is already in production, you need to be more considerate and careful of when and where you perform your testing. This could mean testing during off hours

(often in the middle of the night) or over a holiday period. It could also mean testing in the middle of the work week, starting at 9:00 a.m., on a nonproduction version of the system (pre-prod or staging) so you have people on hand to fix issues as they are found. No matter where you do this type of testing, you should back up your app and data before you start.

NOTE Hiring a good and fair penetration tester can be more complicated than you may realize. Figuring out if someone has the skills you need, rather than someone who is faking it, can be difficult if you do not have those skills yourself. Even if someone has the skills, it can be difficult to ensure that they will do the work they say they will, as some will run a few automated scans and fudge their contract numbers, thinking you won't know the difference. This is sadly much more common than you may realize, and I caution you to ask for recommendations and/or references for good firms or individual testers before you shell out your hard-earned cash.

ALICE'S ETHICS

Alice went on a date with a penetration tester a couple of years ago. After they had gotten to know each other over a few glasses of wine, he proudly explained that he was “faster” than other testers. Alice said that seemed like a disadvantage because he got paid by the day, and if he ended his contracts early, wouldn't he get paid less? What's the advantage of being faster? He explained that he still charged for the full 10 days of his contracts, but it only took him 3 or 4 days' worth of work. Alice was shocked! “But that's fraud,” she said, as her date attempted to backpedal. They didn't go on any more dates.

TIP With *coverage-based* testing, the tester will look for as many ways as possible that they can exploit your system. *Depth-based* testing means they will test one part of the system exhaustively. Choose which approach you want with the assistance of your security team. Coverage-based testing applies significantly more often than depth-based testing.

TIP If you are shopping around to hire someone to do security testing for you and the price is very low, be wary. If they are hiring skilled people from countries with weak currencies, that might still land you a good result, but it could also be that they are just running automated scans and *pretending* to do manual testing. Ask how they plan to test the business logic of your application and if threat modeling will be included in the engagement. Ask what techniques they plan to use and if they are testing to OWASP's ASVS (Application Security Verification Standard), the OWASP Mobile Application Security Testing Guide (MASTG), or another framework. Ask what tools they are using and how they validate their test results, and request a sample report to see if you can understand what you will receive at the end and what methodology(ies) they will follow when testing. If they do not have answers to these questions that make you feel confident that they will do a good job, perhaps you should keep looking.

TIP Get everything in writing if you are going to do a potentially destructive test (note: penetration tests, stress tests, performance, and Denial of Service (DoS) tests are all potentially destructive). Although I generally try not to prejudge most situations, even though I've never been sued before, I feel confident that I wouldn't like it, and I suspect that you wouldn't either. Let's avoid that possibility by thoroughly documenting the scope of testing, types of testing, tools you will use, time frame for testing, IP addresses, and URLs, and make sure you have that magic signature from someone with the authority to approve the test. Once the test is done, keep that signed contract for several years, just in case.

Manual Testing

The difference between an expert penetration tester and running an automated scanning tool is night and day. Although automated dynamic scanning tools can find all sorts of low-hanging fruit, a skilled penetration tester will find everything an automated tool will find and much more. They will test for your threat models. They will test the business logic of your application. They will test what each different user can and cannot do in every part of the system. They accomplish this through manual testing.

Manual testing usually means using a web browser, a web proxy tool (such as Burp Suite or Zed Attack Proxy, aka Zap), and, if APIs are involved, a tool to send and receive requests to and from the API (such as Postman, HURL, http-tanker, etc.). Ideally, the web proxy tool will contain functionality that allows you to repeat requests (so the tester can adjust requests until they have the intended result) as well as the ability to apply automated scripts to run against the application or API (for brute-force attempts, testing input validation, fuzzing, etc.).

When an expert penetration tester (as opposed to an amateur) performs manual testing, they will have a plan. They will have a framework that they follow and specific tests they always perform for all web apps and APIs, as well as specific tests based on whatever programming language or framework your app is built with. They will ask you about your application, your threat models (if you have them), your architecture, and your documentation. They will then form a comprehensive plan to test your application as thoroughly as they know how or that the time in the contract will allow. They will often use automated tools for the parts they can and perform manual testing for the rest. They will then produce a report that explains what they have found, the risk each finding poses, and guidance on how to fix each one.

An amateur will usually only use automated tools. If they perform manual testing, it will often be guesswork, sloppy, and potentially dangerous to your

application and/or the environment that hosts it. If they do find a vulnerability, they may not be able to explain it or offer advice on how to fix it.

An amateur penetration tester, if performing work without the supervision of someone who is skilled in this area, can potentially cause serious damage to your systems. I know this from personal experience, as I, unfortunately, grossly damaged a client's applications during my brief (1.5 years) time as a pentester. I remember being excited about my first penetration test. My first InfoSec professional mentor had me read half of *The Shell Coders Handbook* and half of *The Web App Hacker's Handbook* and told me to learn Burp Suite over the weekend from YouTube videos as my "training." He then let me loose on a client's live production system with zero guidance (he told me he would supervise me, but he did not). I somehow managed to find server-side request forgery (SSRF) and accidentally made a terrible mess of their database and web server. In my defense, I had no idea as I continued to test that I was overwriting files on a live server and that I was not being supervised. I had no idea that any of my attempted attacks had "worked" until it was far, far too late. Although the client was able to recover from backup, they were extremely upset.

NOTE You may notice that I never mention the name of my first professional InfoSec mentor. This is because the lessons I learned from him were mostly "what not to do." He taught me a lot about ethics, but not in a good way. We did not last long, and since then, I have found other amazing professional mentors who taught me about the law, ethics, and how to avoid breaking things by accident. Thank you, Adrien (RIP), Sherif, Nicole, and others for showing me the light.

All this is to say that you should be very, very careful whom you hire. His client didn't hire me, they hired him, and he subcontracted the work to me. At the time, I thought they knew I was doing the work, but after he yelled at me, it became clear that my involvement was not known to the client as I had been told. This meant not only was he in breach of contract, but both of us broke the law (me for testing a system I did not have legal permission to test, and him for giving me access). Even though he had signed a contract with me, I still could have gotten into hot water. It never occurred to me that he would lie to me, especially not in writing. My entire career could have gone very, very differently if the client had taken him to court or called the police.

That said, the client restored from backup, and all was not lost. No data was stolen or destroyed; it just ruined someone's day and wasted their time.

Please get references and recommendations before you hire a penetration tester. People who know what they are doing have clients who will happily sing their praises. If you can't find someone or aren't sure, email me, and I will recommend someone I trust: Tanya@SheHacksPurple.ca.

BOB'S ETHICS

When Bob was in his “I’m going to be a pentester” phase, he met many amazing hackers, some of whom became his friends. He looked up to the people at the Capture the Flag (CTF) contests and meetups, especially the people who had won contracts instead of him. Over time, he got close to someone who was very well-known in the industry. Bob went to his house to hang out and hack one day. At the front door, Bob saw a gigantic number of badges hanging from a hook. Bob asked what that was about: “When you finish a contract, don’t they ask for their badges back? I take it you’re a collector.” The guy proudly explained that he had 35 badges for his active contracts. Bob was astounded! How on earth could this guy manage that much work as one person? The guy then explained that he would go in, start his scanner, and then disappear and go to the next workplace. He would hit several places per day, gather the scan results later, and get his assistant to write up reports by copying and pasting them into Word docs. He had a devious smile when he explained that he charged for a full day of work to start a scan, another when he had copied the scan results, and another for any other activity he had done. He laughed that he could “charge for 8 days of work per day, if he played his cards right.” He then made fun of his clients and called them idiots. Bob was disgusted! He decided right there that he would find more ethical people to learn from, even if this guy was the most popular person in his town. There had to be someone better out there.

Automated Testing

Automated security testing may have changed since I joined the field of AppSec, but I still find it fun. Running an automated scanning tool like a DAST or an API dynamic scanner and seeing it find something still sparks joy for me. DAST scanners tend to be good with monolithic applications, finding low-hanging fruit quite quickly, but they are not as good when it comes to APIs and not very good at all when it comes to single-page apps (SPAs) that are built only in JavaScript. API-specific dynamic scanners are better for finding bugs in APIs. I still haven’t found an automated scanner that works well on SPAs.

Dynamic scanners work by proxying your web app or API and then watching as you use the app. They record the requests and responses and then send their own requests, adjusting based on the responses they receive, attempting to find vulnerabilities. As mentioned before, they are great at finding low-hanging fruit (obvious vulnerabilities) but not as good at finding more complex issues and terrible at finding business logic flaws. This is, of course, my professional opinion as of this writing. I feel confident that anyone who sells a DAST will tell you something different. Either way, I think running a DAST scan and fixing what you find before you go live is a worthy activity. That said, if you are going to run a dynamic scanner, I suggest that either you get a professional to do it or you do it yourself but in a nonproduction environment, with nonproduction

data, as they can occasionally cause damage. No matter what, ensure that you have the permission of your security team before you run one, as it's considered "hacking," and no one wants to cause a security incident by accident.

TIP With AI changing literally everything recently, it is my hope that DAST vendors are able to add AI to their tools to make them truly brilliant. I would be very happy to see a dynamic automated scanner that reliably finds logic flaws like pen testers do. One day!

As a software developer (and by that, I mean a very skilled and technical individual), you are more than capable of running a DAST against your apps; it's easy! You will need permission from your security team (in writing) and a safe place to do it (like a virtual sandbox area or your development server). You will then need to pick your tool (Zap is free, so perhaps start there) and point it at your web app and wait for the results to roll in. The first time I did this, I found many bugs and felt very powerful! Try it and see for yourself.

BOB FINDS A ROLE MODEL

After a few months of meeting people at meetups, Bob met an amazing person named Sherif. Although the guy wasn't loud and didn't brag like some of the other pen testers, Bob could tell right away that this guy really knew his stuff. Slowly, Sherif took Bob under his wing, offering career advice, opportunities to learn, and lessons on how to be ethical but still make money. Bob felt like this guy cared a lot about doing right by his clients and making the world a more secure place. Bob decided he had found a new role model.

Fuzzing

Fuzzing is a feature that is often available from within a DAST or web proxy tool. It tests the input validation of a field by entering bad data into the field to see how it reacts. I remember watching Burp Suite start with a single letter "A" in a field, then 50 of the letter A, and then 5,000. It entered all sorts of things until the app responded slightly differently, and it alerted me. I dug into that field and realized the input validation had a small mistake—And that is where I found a vulnerability!

Ideally, as you learned earlier in this book, input validation should always use an "allowlist" approach rather than a block list. A fuzzer can sniff out a blocklist in two seconds! It will figure out which characters you missed in your blocklist via trial and error, and then you're in trouble. It does it quickly and in an automated fashion. Then, a skilled penetration tester figures out how to use that list of characters to get past your defenses.

Fuzzers are our friends, though, because they help show us when we've made a mistake in our input validation. They are generally included as a feature in most dynamic scanning tools, and you should take advantage of them by testing all of your input fields.

LETTING GO IS HARD

When Bob did his first pentest on his own, he was determined to work all the hours of his contract and do a really great job. He was going to be like Sherif by being thorough and diligent. He not only managed to do quite a bit of security testing but also did a threat model, helped his client rearchitect the app so it was more secure, and even gave them a mini plan of how to ensure that all their apps were more secure from then on. At the end of the contract, it was hard for Bob to let go. He felt his work was not yet done, even if he did as much as he could in the time frame allotted.

Interactive Application Security Testing (IAST)

IAST is a newer technology (about a decade old) that tests your application as it is in use, from the inside out. It's installed inside your application as a binary, a process called *instrumentation*. IAST needs to be compatible with your language and framework, meaning if you are using something obscure, it likely won't work. IAST tools only work as your application is being used, meaning if you install it on your development server and hardly ever interact with that instance of your app, it is unlikely to find very many bugs there. That said, if you install it in production and have it running during a penetration test, it is likely to find quite a few interesting bugs for you.

Although I have never had the pleasure of using an IAST tool in production where I work, I have gathered a lot of feedback from clients over the years about this type of product because I'm very curious to know what I've been missing out on.

Feedback I've received over the years about IAST tools:

- They are quite pricy when compared to other AppSec tools.
- They provide a high percentage of true positives and a low percentage of false positives.
- They cause a very small amount of latency.
- They take quite a bit of engineering time to install—time that could be spent fixing bugs. If you have a huge backlog, you might not be in a place where IAST tool makes sense for you.
- They tend to give value later in the SDLC, as you cannot get good results out of them unless you have a lot of usage of your app. You cannot shift an IAST tool left.

- Almost all of my customers that used IAST tool were banks or some other kind of financial institution.
- They are generally used in very mature AppSec programs and only on the most important application(s) in their programs.

Very few of my clients use IAST tool (I would estimate 1%); however, the ones that do seem satisfied. I will let you be the judge and hope this information I have gathered over the years is helpful.

Bug Bounty Programs

Another way to perform testing is to run a bug bounty program. Moreover, it's a way to outsource a portion of your testing to security researchers. An organization that runs a bounty program invites security researchers (also known as *ethical hackers* or *bounty hunters*) to test a safe-to-test copy of one or more of its systems. Bounty hunters receive payment for any bugs they submit that are (1) within the scope of the program, (2) exploitable (not just a scanner result—the bug must create real business risk), and (3) not previously reported or otherwise considered “a known issue.” The company must clearly define the scope in advance, just as it would for a penetration test, and it should also include how much is paid for each type of bug, how to submit, how to apply for the program, and other pertinent details.

If you decide to do one of these programs, prepare for *a lot* of reports. Not all of the reports will be great; there may be quite a lot of noise (false positives, reports that make no sense, poorly written reports, reports without enough information, bugs that are out of scope, etc.). That said, you may also be impressed with the amazing things bounty hunters find. They can be truly ingenious and creative in the way they find bugs.

TIP Sometimes, inexperienced or unethical bounty hunters will attack your customers, take over their accounts, and then try to ransom them back to you (demanding a bounty to release the account back to your customer), all while pretending they are “a legit bounty hunter.” This type of vulnerability they are attempting to report is called *account takeover*. Account takeover is almost never a part of a bounty program's scope, and no organization will ever condone a bounty hunter attacking one of its customers. Also, live customer accounts are not generally in the scope of a bounty program. Please include clear and pronounced language that account takeover is always out of scope and that if any bounty hunter attacks one of your customers (in any way), they will be banned from your program and reported to the local authorities. This may save you and your customers some headaches.

It is my opinion that you should have a good AppSec program before you even consider starting a bounty program. I don't believe it's a good use of

anyone's budget to pay \$500 to find an obvious cross-site scripting (XSS) bug that you could have found yourself in 10 minutes with Zap or Burp Suite. You should already have scanned your apps with multiple types of tools and fixed everything of concern before you even consider setting bounty hunters on it. You should also make sure that you have the resources available to validate the results that are submitted via your program (there will be a huge number of reports to sift through) and also that software development time is allotted to fix the issues that will be found. If you do not have all this ready before you begin, you will not maximize the value of your new program.

Some vendors may try to tell you that you should begin a new AppSec program with a bounty program, saying "it's a great first step." I vehemently disagree. If you are not sure whether you are ready to start a program, hire a professional consultant to help you evaluate your readiness, ask a trusted advisor or mentor, or ask people from your InfoSec community of choice for advice. It's okay to ask for advice, and this is a super-interesting topic that I'm sure any colleague would be enthusiastic to discuss with you. Seriously!

NOTE I have strong feelings about bug bounty programs. It is my opinion that if high-quality researchers are participating, plus you have great oversight and management of your program, this can be the absolute most intense and amazing testing you can do. While I was at Microsoft, some of the bugs reported to its bounty program were incredible. But I also believe that if you let just any person submit bugs to your program, you will drown in terrible bug reports that offer little value to you and your team. I also believe that if the program is not well managed, you could end up paying a lot more than you need to and not get the results you desire. This is easy to get wrong, and you need to do a lot of prep work before you launch your program to get it right. If none of your team has experience running one of these programs, I strongly advise that you consider bringing on outside expertise to help you get started. I personally recommend Luta Security, but there are also other well-known companies such as HackerOne, Bug Crowd, Synack, and others that serve our industry.

Test Results

Every security team is different, and that means they all deliver test results in different ways and formats. I hope your security team is helpful, generous with their time, and informative in how they present their findings to you and your team. But that might not be the case at every workplace.

Ideally, you would receive security test results in the same place where you receive all your bugs: in your bug tracker. Each item would contain clear and concise information on how to reproduce the bug, an easy-to-understand explanation of the risk it poses, advice on how to fix it (often called mitigation advice),

and maybe even code samples. If you're really lucky, you might even receive a pull request with a suggested fix (even better, the fix is actually good and doesn't break your beautiful app). Hopefully, there's information included that helps you decide the urgency with which you should treat this issue, and all the items are prioritized. Last, if all is right in the world, someone (or *something*, such as an AI) has triaged the results before you ever even see them, to ensure that all the results are valid, true positives so you do not waste any time on a wild goose chase, tracking down something that isn't a real issue. I hope this is your experience wherever you work.

That said, for many organizations, this is not how security findings are received. You may receive test results in any number of the following ways:

- In a formal presentation, with slides presented by a penetration tester, in a meeting that probably has a lot of managers in it.
- In a PDF, emailed to you with no other information.
- Your Continuous Integration Server (CI) breaks the build, and then you sort through the logs to find that a security tool thinks it found a bug, and you are not going to prod until you fix it (or suppress the result).
- You have a meeting or receive an email where you're asked to log in to a security dashboard, and then you have to sift through everyone else's bugs to find yours.
- Someone shows you in a spreadsheet.
- One of a myriad of other inconvenient ways that do not work with your regular development processes.

If you receive bugs from the security team in a way that you find less than ideal, I suggest telling them how you normally receive bugs and then offering to help them automate their tools to deliver them that way.

Actioning Test Results

Let's say that you have received a list of security issues for one of the applications you are charged with building or maintaining. Hopefully, you are given time in your project, or by your manager, to action these issues (if not, ask them). With this time, you will want to prioritize what to fix first. If you're lucky, the bugs are already prioritized! If not, you may need to ask for advice, or you can read the description and judge for yourself which ones seem the worst and action those first.

Read the mitigation advice and see if it makes sense to you. If so, great! Run off and fix the issue. If not, you might want to see if it's mentioned in this book, flip to that section, and go from there. If that fails, I suggest seeing if there is an OWASP Cheat Sheet to help you by searching for it online. If there's no cheat

sheet, just search “OWASP” and then the name of the vulnerability, and go from there. OWASP almost always has good advice to share.

Once you have a fix that doesn’t break anything else, if you have access to the tool that found the bug, run it, and see if you pass now. If so, *awesome*. Submit the fix to be merged into the branch you’re working on. If not, try again until you do.

If you do not have access to the tool that found the results, submit the fix however you normally do it, cross your fingers, and hope it passes. It’s better if you can test it yourself, but sometimes, we don’t have that option.

I realize that I am simplifying greatly here. You may need to do quite a bit of debugging, especially if the issue spans multiple files (known as *cross-file* or *interfile*). It takes time, patience, and often also caffeine or sometimes even a bit of swearing. But you can do it. And it is worth the effort! I thank you for doing it. *Developers who fix security bugs are unsung heroes.*

NOTE A significant percentage of the security industry believes that software developers don’t care about security, and the reason is the large number of security bugs in the backlog. They believe that software developers *don’t want* to fix security bugs. According to those folks, devs don’t fix them fast enough, or they don’t fix enough of them, or they don’t fix any at all. Personally, I do not believe software developers don’t care about security. What I believe is that programmers have many priorities and things they care about, and security is one of those things (but not the *only* thing). I believe they are often not given enough time, or any time at all, to fix security bugs. On top of that, at many organizations, they are incentivized *not* to fix security issues because they only receive bonuses, promotions, and raises for new features and never for fixing security issues (I consider this a “perverse incentive”). As a result, security teams are often frustrated with developers, and I feel these frustrations should be focused on management instead. But I digress.

WARNING Please don’t suppress bugs or mark them as false positives in security tools if they aren’t truly false positives. If you aren’t going to fix something, whether due to lack of time/other priorities/whatever, mark it to say that instead. *If you suppress a valid result as a false positive, you are causing harm.* First of all, you’re teaching the tool the wrong thing (so it might miss other issues in the future, especially if AI is involved). Second, you’re skewing the metrics so that the tool looks bad, meaning the security team may incorrectly think it’s not working properly. Third, and most importantly, the organization doesn’t understand that app’s current level of risk. If you suppress 20 results because you know you can’t fix them, everyone will think the application is safe when it’s not. It’s better to know it is not safe than to hide it from your organization. The security team also isn’t able to argue for more resources if it looks as though all the bugs are getting fixed. I would much rather have accurate information, even if “it looks bad,” than misunderstand the security posture I’m expected to defend. Please, as a favor to me, do not mark things as false positives if you know they are real bugs. Thank you.

Final Thoughts

If we do not test our systems, we do not know the risk they present to our organizations, which leaves us unprepared in our defenses. Without testing, we cannot be sure our mitigations, bug fixes, and defenses are functioning properly and that we are indeed secure. Whether you are the person who writes or designs the software, the tester, the security engineer, or any other job in between, it is part of your duty to ensure that the systems you build and maintain are safe for your users, customers, colleagues, citizens, and loved ones. Testing isn't about finding more bugs and then having more work; it's about ensuring that we build high-quality systems, and we cannot have high-quality systems that are insecure.

Chapter Exercises

You can find the answers to these questions in Appendix B, "Answer Keys."

1. In your own words, why is scoping an important part of any security testing engagement?
2. Why do we need to get a signed document before we start penetration testing?
3. In your own words, how is penetration testing different from running an automated dynamic scan?
4. Dynamic scans are potentially dangerous. True or false?
5. Which provides the highest-quality test results: automated testing only, or a combination of both manual and automated testing?
6. The purpose of fuzzing tests is to validate that we are using parameterized queries. True or false?
7. In your own words, why is it important that our penetration testers have good ethics?
8. In your own words, why is it important that you, as an IT professional, no matter what job you do, have good ethics?

Release/Deployment

During the act of deployment, we can continue to ensure that your applications are secure, thanks to the invention of continuous integration and continuous delivery (CI/CD) systems, which automate the release process for us. In this chapter, we will discuss adding security assurances to those systems, as well as how to secure the CI/CD system itself.

Whether or not you have manual verifications or approvals in your CI/CD, it is still usually a huge improvement over a manual deployment process.

TIP If you are not using a CI/CD or build server to release your applications, all is not lost! You can still do a great job of securing the software you build. You can perform testing before (static application security testing [SAST], software composition analysis [SCA], secrets, linting) and after (dynamic testing or interactive application security testing [IAST]), you compile and deploy your application. You can then manually create a checksum of the compiled artifact to save into your code library as evidence in case you need to prove the integrity of your build later. We will cover how to harden and monitor the CI system later in this chapter, as well as your applications (next chapter), to ensure that no one attempts to exchange your copy of the application for a modified version.

SECURITY HYGIENE TO THE RESCUE

Back in the day, Bob used to work with a development team that rolled out everything manually. He was shocked to find out that it took them 2 weeks of effort from the entire 18-person team to roll out a new version of their app, which they did every 3 months as part of their “mini-waterfall” approach to software development. Every single time they deployed, there were problems. QA didn’t match production, staging crashed, UAT’s (User Accept Testing) database wouldn’t connect, and it was beyond stressful. Eventually, Bob figured out that the database administrator (DBA) had been manually coding the scripts for each environment and that they were not tested in advance. He also found out that there was no rollback script, so whenever the DBA made a mistake, he would “deal with it,” which was often the cause of the delays. Bob instituted the same formal processes for the database changes as for the rest of the code, with the use of source control, testing, etc., during the following 90 days, despite constant complaints and being told he was “overreacting” and “being a perfectionist.” The next mini-waterfall release took 3 days, with no major errors. During every release from then on, for the rest of the project, Bob focused on accuracy, testing, and reproducibility and wouldn’t let anyone do anything manually. Every new release took less time, with fewer problems. Bob might not have been the most popular person on the team at the start of the project, but he took that project from “wildly behind schedule” to “delivered with plenty of time” in one year. Bob was very proud of his team and the changes they made together.

Security Events Within the CI/CD

A CI/CD pipeline is meant to go *fast*. It is meant to take not hours but minutes. When we add security testing to this type of system, it is extremely important that the testing we do is fast, accurate, and reliable. This means that if your tooling is known to produce false positives, it is probably better to perform that work outside of the automated release process. It also means that if the test will be quite slow, you might want to automate it: for instance, to run on an overnight schedule instead of in the pipeline. I have seen well-intentioned security teams many times add a first-generation SAST tool to a CI/CD system, only to have it cause countless alerts and break the build on false positives, causing development teams to become extremely frustrated. This creates friction, wastes time and money, and often ends up with the security tests disabled. Learn from previous mistakes, and be careful to thoroughly test a tool before you add it to a CI. Then show it to some developers, get feedback, and adjust accordingly.

You can perform pretty much any type of automated testing in the CI, but not all are advised. I suggest avoiding or limiting the following types of tests:

- Dynamic automated scanners (dynamic application security testing [DAST]) tend to take a very long time, especially if you need to do cataloging/

spidering/figuring out every link of the application. Instead, consider either running the scanner on a schedule outside the pipeline, only performing passive scans (analysis of requests and responses, without fuzzing or sending any requests itself), or only running extremely important tests, such as authentication, injection, etc., and turning off the other tests.

- Fuzzing can be quite slow, and sometimes it can make a bit of a mess. You will want to ensure that you fuzz against test data (or backed-up data), and you may prefer to do this type of testing outside the pipeline.
- First-generation static analysis, if the tool performs symbolic execution that can take a long time and provide quite a few false positives. Instead, consider running it outside the pipeline or only running certain types of tests such as injection, XSS, and other critical vulnerability tests. Second-generation SAST is usually fine in a pipeline (antipattern matching).
- Stress and performance tests should always be done outside the pipeline, full stop. You want to have all resources available to you during these types of tests to know your full capabilities. For further details on stress and performance testing, see the testing chapter in *Alice and Bob Learn Application Security*.

Breaking the Build

I have worked with many security teams that look forward to “breaking the build” with the belief that it will change developer behavior, ensure that they have more secure applications, and solve a lot of problems. Unfortunately, breaking the build is not a magic bullet.

Breaking the build means stopping the CI/CD from releasing any further (no matter which stage you are in) after a test fails, and you generally do this to avoid a very serious vulnerability or bug from accidentally being released into production. We do not break a build for minor reasons or for just any old vulnerability. We do not do it to punish or embarrass anyone. It is an emergency stopgap, similar to pulling the Andon cord on a physical assembly line. If you break builds all day long, *you are doing it wrong*.

Ideally, you should have a system development life cycle (SDLC) that includes multiple security activities that happen before the CI/CD starts, and the code you are attempting to release is in pretty good shape before you start the CI. The checks in the CI/CD are in case you missed something earlier. They are not your first look at security; they are often your very last check before going live.

TIP When implementing a new tool into your pipeline, you should always begin in alerting/monitoring mode. One day you may graduate to blocking, but never start there. This way, you can observe the tool in a real-life scenario without wreaking havoc on the rest of the nicely tuned tools already in that CI.

TIP If you are breaking builds constantly, not only is that not very efficient, but also you are likely not making very many friends. Try to “shift security left” by starting your testing earlier in the SDLC. Security tools belong in the developer’s hands too, not just the security team’s.

Secret Scanning

Secret scanners look for secrets in your code. Ideally, as a first step, you scan your entire code base and remove the secrets so they are accessed programmatically from a secret management tool. Then you create a pre-commit hook to scan your code as you check it into your code repository and block you from checking in if it finds a secret. That way, you can avoid a security incident (checking the secret into source control means you’ve spilt it, and it’s too late to go back at that point—you have to rotate it and start again). If you have a pre-commit hook, there’s no need to also scan for it in the CI/CD, assuming your CI always takes code directly from your repo and you’re not doing anything bizarre, such as copying code from somewhere other than your code repository to run through your CI.

Static Analysis

Second-generation SAST tools perform antipattern matching, and they can go quite fast (think seconds to minutes rather than hours to days). They are designed specifically to work in a CI or IDE type of setting and tend to have mostly true positives. That said, sometimes they have false negatives (they can miss things), so please ensure that you do more than just one type of testing. There is no single type of security tool, as of this writing, that can do every type of test you need.

Dynamic Analysis

As stated earlier, I’m not a huge fan of putting a DAST tool in a CI/CD because it’s *a lot of work*. You can prerecord your cataloging of the site via a HAR (HTML archive) file or cut down the scope to something tiny (only test for 10 types of vulnerabilities, or only test an extremely small section of the app), but it’s still slow, and those results will not be very helpful. Traditional DAST also tends to report quite a few false positives and negatives, meaning if you are in blocking rather than alerting mode, you might be falsely breaking builds. I still want a DAST tool, but generally I only run them in passive mode in a CI (looking at requests and responses but never sending their own requests) if I have them in the CI at all. I prefer to run a DAST tool outside the pipeline either manually (for thorough testing, which includes manual testing) or automated overnight/off-hour scans on “full blast” (I test for everything, knowing it will take a long time, and I’m cool with that). I then examine the results later, at my leisure.

Software Composition Analysis

SCA is great for a CI/CD, as it's very fast. As stated before, I prefer an SCA tool that can tell you if the vulnerability in your library is reachable or exploitable, and I hope that soon, every SCA tool can do that instead of just some of them. I would always put an SCA tool in CI, assuming I have licenses available.

Linting

Although linting is technically more about quality than security, I still think they are great tools. I prefer to do my linting in the IDE because I'm a control freak and want to fix everything before anyone else sets eyes on my work. I feel that way about every tool, even if that's not necessarily reasonable or always a great idea. Linters are also a great choice to put in your CI and can be implemented as a pre-commit or pre-merge check, whichever works best for you!

Infrastructure as Code scanners

Even if we're technically only in charge of the applications, we want to ensure that the infrastructure that serves our apps is in a decent state. If we are using infrastructure as code (IaC) to build that infrastructure, we can use an IaC scanner to validate that we have made secure choices along the way. They are fast, tend to be quite accurate (compared to SAST and DAST), and offer actionable advice (patch this, change that config, etc.).

TIP Examples of IaC include Terraform, Azure ARM templates, and CloudFormation.

Unlike a vulnerability assessment (VA) scanner, an IaC scanner does not need a running container or virtual machine to analyze; it can read whatever language is used in your IaC file (often YAML).

P.S.: Make sure you handle your IaC files the same way you would any code; check it in your code repository, perform version control, test it thoroughly, etc.

Securing the CI/CD Pipeline Itself

This year, OWASP released even more "Top Ten" lists, and one of the new ones is called "The CI/CD Top Ten." It is a list of the top 10 risks to CI/CD systems, and I will cover several of the items OWASP lists in this section. The list is great, and you should check it out: <https://owasp.org/www-project-top-10-ci-cd-security-risks/>.

As discussed previously in this book and many DevOps books, a CI/CD has incredible power and privilege; if controlled by the wrong hands, it can be

abused to cause untold damage. It is critical that access, confidentiality, integrity, and availability of these systems are strictly controlled and protected.

The first thing to address with a CI/CD is access control: who can change what and keeping track of the changes (also known as authentication, authorization, logging, and auditing; OWASP #1, #2, and #10).²⁸⁹ You do not want someone who is not trained properly to be able to make changes or release code without proper approvals, just the same as someone who has bad intentions; they can both cause huge problems for you, your organization, and your apps. Lock down access control, turn on auditing and monitoring, and validate often that only those who should have access do. When someone leaves your org, make sure access is removed. Ensure that you have the proper approvals as per any organizational policies built into the system as technical safeguards (forcibly stopping the build until approval is given). On top of this, do not allow employees to use their personal email addresses to access work-related systems, including the CI.²⁸⁹

The second thing to look at with a CI/CD is the integrity of the packages and dependencies you are using (OWASP #3). Often we are calling from various places for all the third-party pieces we need to build our app, but we are not validating checksums or signatures, or we have been confused in some way into downloading the wrong versions. Validating that we are indeed using the correct versions of each piece every time is becoming even more important with every new successful software supply chain attack. Also, do not pull images or packages from untrusted sources, such as Docker Hub, where anyone can put anything up there to be shared. Get your pieces from trusted sources. If your cloud provider offers automated vulnerability scanning of your images as part of the package, take advantage! Audit your complete flow of packages, images, and anything else you include in your project to ensure that there is an integrity check for each item.

TIP Docker Hub, GitHub, Stack Overflow, and many other online sharing platforms are great places for sharing ideas and learning. That said, be aware that any online sharing platform that allows the public to upload code/images/libraries/anything means the content you find there is potentially dangerous and should be considered untrustworthy until proven otherwise. Proceed with extreme caution when using anything from such a site in a work environment, and follow any applicable work policies as well.

Ensure that all tests, scripts, make files, and anything that is executed as part of the pipeline is only accessible to trusted individuals, has an integrity check before it runs, and is managed in source control.²⁸⁹ Ideally, these files are located separately from the rest of the project code, as there is a strong likelihood that the entire team does not *need* to edit these files. Having a subfolder with different identity and access management (IAM) rules can work well in such a situation.

Limit the permissions of the pipeline itself from the rest of your environments. Often, developers will add more permissions than they need to make sure something works, and forget to remove the extra permissions later. If your pipeline runs with a service account that has been given enormous access and rights, and someone malicious gets control of that service account, they could use it for evil. This includes access to secrets! Only give each part of your CI the permissions it truly needs. For instance, perhaps the entire development team can approve releases to the development and QA environments but nothing beyond that, or the team lead can make changes to any part of the CI but the rest of the team can't. Audit the permissions annually or more often.²⁸⁹

If your pipeline is local, not cloud-based, ensure that you have a robust backup plan. I've seen people lose their pipelines, and it took weeks to reconfigure everything so that it was "just right." It's heartbreaking to lose code, and your pipeline is code!

When dealing with secrets, it's important not only to protect them from getting into your code but also how they are accessed and used within the CI/CD system. Ensure that they aren't leaking (such as being saved into the CI logs or shown in error messages), being misused, or given open access. If you see that they have been spilled (or even suspect that they have been spilled), rotate them immediately.²⁸⁹

One of the items on every single OWASP Top Ten list is security misconfiguration, and it stands true for CI/CDs just like any other system.²⁸⁹ System inventory, hardening, annual audits, applying least privilege, doing a pen test, etc., should apply to your CI/CD, not just the apps you create. These systems are amazingly powerful, and (literally) locking them down is mandatory. Follow the hardening guide.

Most organizations connect their CI/CD to third-party systems, often Software as a Service (SaaS) products. These products need to be not only locked down but also documented, and you need governance around them. Although it's great to connect a helpful tool to your DevOps pipeline, you need to ensure that it is configured correctly, that it's deployed everywhere it is supposed to be, that you are using the correct number of licenses, that it runs every time you think it runs, that you set policies in the tool that match your stated policies, etc. Even a great tool can be used improperly. You should also have an approval process for adding new tools, a system that allows for visibility of each tool (ideally you are collecting metrics as well), a policy and matching process for when they are decommissioned, and monitoring of usage (both to ensure that you're getting your money's worth and so you don't unknowingly break a license agreement).²⁸⁹

You can sign not only your code but also each artifact produced by your pipeline (such as reports, logs, or images). Doing so and then validating the signatures (or checksums) at each step in the pipeline will also help protect against attacks on your CI system.²⁸⁹

The last item I want to bring up, which is not on the OWASP Top Ten for CI/CD, is uncontrolled resource usage. This can mean someone accidentally making 3,000 images instead of 30 or someone running thousands of pipeline runs with no business requirement for doing so. You need to ensure that there are limits, every opportunity you can, when it comes to technical systems. It is very easy to leave things ambiguous or with no limit at all and then end up with a terrifying cloud bill.

2,365 PIPELINE RUNS A WEEK!

Alice did a contract at a large business that spent 2 years trying to transform itself into a DevOps shop. One of the people showed her all the cool new pipelines they had made. He let Alice click around and see how it all worked. Alice hovered over a pipeline's name with the mouse and saw a name pop up; Alice asked what it meant. The person giving her the tour said that pipeline belonged to him. Then Alice clicked the person's name, and a long page with a list of many pipelines came up. It said, "Page 1 of 133 pages." Alice said, "But that means that guy has thousands of pipelines. Why does he have so many? How many are people supposed to have? Doesn't that seem like a lot?" The man giving her the tour made a face. After a few hours of investigation, it turned out that Alice had uncovered an employee who had been making pipelines for fun, for over a year, on the company's dime. His 2,365 pipelines ran every week, costing the company thousands upon thousands of dollars. That employee was let go, and the company instituted strict auditing on its systems from then on.

Assuring the Integrity of Your Release

I realize we just covered a lot of this topic in the previous section, but I wanted to dive a little deeper into this specific part of the SDLC. Whether you are doing DevOps, Agile, Waterfall, or something else, being able to prove the integrity of the software you release can help you create trust with your customers and your security team. It also helps prevent security incidents! Following are a few things you can do to ensure that (1) what you are releasing is what you think you are releasing, and (2) you can prove it.

- Digitally sign your software releases and/or data packages. Especially serialized data.²⁹⁰
- Ensure that libraries and third-party dependencies are from trusted repositories. Consider hosting an internal repository of "known good" (vetted) components.²⁹⁰
- Ensure that a software supply chain security tool (also known as SCA) is used to verify that components do not contain known vulnerabilities and, if they do, that those vulnerabilities are not exploitable from within your

application. If they are, upgrade off. Treat it as you would any vulnerability in your code; fix it.²⁹⁰

- Perform code reviews. Look for both malicious code *and* vulnerabilities.²⁹⁰
- Create a review process for configuration changes to your IDEs, code repositories, sandboxes, pipelines, or anything else you use to create your software.²⁹⁰
- Follow the advice in the previous section of this book.
- Do not send or accept serialized data to or from an untrusted source. If you are receiving it from a trusted source, or sending it, ensure that it is encrypted and digitally signed. Validate the signature before deserializing.²⁹⁰

SURPRISING CLOUD BILLS

Alice's colleague hired a co-op student (intern) last semester who was a total whiz kid. Her colleague was so impressed that he gave the kid free rein over the CI/CD and let him release to production without approval after only a couple of weeks. Alice walked into the cafeteria a few weeks later and saw her colleague, his head in his hands. "What's wrong?" asked Alice. He explained that the whiz kid had accidentally added two extra zeros before deploying to Kubernetes, and he had just received a "surprise" \$30,000 cloud bill! He didn't know how he was going to tell his boss; his budget was already in the red. Oh, no.

Security Release Approval

Quite often, the security team will want to review things before they are released into production. Sometimes, this requires multiple steps, paperwork, and meetings; and other times, it's just a brief discussion or a few emails. It all depends on your security team's policies, your organization's appetite for risk, legislation and compliance requirements, and how many security people you have (sometimes they would prefer a lengthier or more in-depth process, but there just isn't enough staff to make it happen). If you want to get your code out into production as fast as possible, I suggest learning what steps are required to get approval and starting them as soon as you can during your SDLC. If you need to perform a threat model, get it scheduled. If you have to pass a static analysis scan, see if you can get access to the tool so you can scan it yourself first and fix anything you find; then you will pass the security team's scan with flying colors. If you need to fill out a survey or a security architecture document, start on it right away. Although this is likely not your favorite part of the project, starting early will mean you aren't slowed down later.

Chapter Exercises

You can find the answers to these questions in Appendix B, “Answer Keys.”

1. Why do we need to secure our CI/CD? What’s the purpose of that? What could go wrong?
2. What is the purpose of digitally signing your code? What does it achieve?
3. Why would we need to validate the integrity of a third-party library? What could go wrong if we download something other than what we intended and then publish it as part of our software?
4. Why is access control important for a CI/CD?
5. Name two types of testing that you should put into the CI/CD.
6. Name one type of testing you should not put into the CI/CD.
7. Name your favorite type of security testing. Why is it your favorite?

Maintenance

Quite often, when working with companies that are not startups, I find that the majority of the security attention is paid to the new apps, not the legacy or preexisting applications. I often have to spend a lot of time advocating for the allocation of time, resources, and money to spend on securing legacy applications. I find this rather confusing because if you look at the security posture of the software across any mature (older than 5 years) organization, it is always the legacy applications that create the majority of the business risk, not the brand-new ones. I don't mean there are more legacy apps than new ones, and, therefore, there are more bugs. I mean that software ages badly from a security perspective. Each year that goes by, our industry takes security more seriously, and therefore, we are creating more secure apps, most programming frameworks and languages become more secure with each new release, and everyone is becoming more knowledgeable about security as we continue to have breaches, and it is less likely that old applications have had thorough testing when compared to new ones. On top of this, the longer software exists, the more time malicious actors have to find security issues, and this includes not only the products we have built but also the libraries and other components we used to build them. This means that even if we built it perfectly way back in the day, the pieces we built it with might have degraded. All these factors add up to older software usually being less secure than newer software unless your organization prioritizes the security of legacy systems, and I have found that to be rare.

When working in AppSec, I have always tried to spend half my time working on securing legacy systems, despite the unpopularity of that stance at some of my workplaces. I urge you to consider allocating your security time this way as well, if possible. I realize that you don't always get to decide what you work on, but if you have some sway, the absolute oldest applications where you work might be the place where you can make the largest possible positive impact.

In this chapter, we are going to talk about how to maintain a system so it ages *gracefully*. We want our software to continue to serve the business for the long term in a safe and consistent way. To ensure that is the case, we have some work to do!

NOTE In this book, I will refer to *legacy systems* when I mean custom software that has been in production for a year or more. Usually, companies do a penetration test and other security activities before releasing to production but then do not continue giving it much attention afterward. This means these systems become less secure over time unless you do proper maintenance. Although the one-year-after-release timeline is completely arbitrary on my part, after one year in prod, I feel that teams absolutely must retest their systems.

ACQUISITION BLUES

Alice's company bought a small business last year that was a competitor. The company bought the competitor because it wanted the competitor's customers, but it also wanted one specific web app product that was more popular than what Alice's company made. Alice's company wanted the features that the customers liked so much. When Alice brought the company into the fold, the security team insisted on scanning the new company's apps for vulnerabilities. They came to her a week later, red-faced. They told her that they had run something they called a SAST against the main app that Alice had bought the company for, and it had *43,000 vulnerabilities!!!* Alice didn't understand how this was possible. She told them their tool must be broken. The security team acknowledged that they were using a first-generation static analysis tool, so there would definitely be quite a few false positives, but they had started validating the results manually and already found 10 instances of injection. They said the app would cost hundreds of thousands of dollars to fix, or they could just rewrite it from scratch. They said it "constituted catastrophic business risk" and that she should have let them evaluate it as part of due diligence. Alice changed the procedures where she worked to always include extensive security testing on important applications from companies her company planned to acquire in the future.

Monitoring, Alerting, and Observability

When we monitor software, we implement systems to observe and track the operations and activities of our users, our applications, our infrastructure, our

network services, and any other systems that interact with our software.²⁹¹ This can mean cloud-hosted web applications used by millions of people or Windows APIs on your host computer used by your internal staff. We monitor software to gather metrics to ensure that everything functions as expected and to help detect any problems, including security problems.²⁹²

A *security alert* is a notification or warning that is triggered when a perceived or overt threat is detected in a computing system or network.²⁹³ We want our custom software to issue alerts when it's in trouble so we can investigate or react to limit damage as quickly as possible. We also want the logs that our software creates to be sent to the SIEM (security information and event management system), if our organization has one, for even more monitoring and alerting beyond what we have programmed into our systems.

There are monitoring products that you can buy and install, and to be quite blunt, they are pretty darn useful (not just for security!). They can help ensure that your apps are up more often, that they run faster, and that you can visualize your performance data in real time. They help you understand the user experience and health of your apps. They can also let you know that you're having a security incident or a regular IT incident. Note: they will not detect every type of incident, but they are still very helpful.

Sometimes, monitoring products are called *observability* products, but there is a difference, despite what the marketing messages may say. *Monitoring* means collecting data and generating reports on different metrics that define system health. *Observability* focuses on investigation, examining system interactions and the data collected by monitoring to try to find the root cause of issues.²⁹⁴ If you can get your hands on an observability tool, on top of monitoring and alerting, it will be very useful. You can ask your security team if they have one and, if so, if you can have access to “observe” the applications under your care. You might be surprised by what it shows you and just how many threats your apps face on a day-to-day basis.

Having all these systems in place will set you apart from and ahead of most other software development shops as a group of people who care about availability, security, and quality above all else.

When it comes to strategy around alerting, observability, and monitoring, we need to ensure that we cover three important aspects:

- *Logs*: Detailed records of events and activities of your system for tracing the history and order of everything that happens in your app.
- *Metrics*: Measurements of your system's performance, such as user transactions, database updates, CPU usage, memory consumption, etc.
- *Traces*: Distributed traces that track the journey of a request or transaction as it passes through different components and services within your application (this includes API calls, serverless, etc.)

Without logging, you cannot properly monitor. Without logging *and* monitoring, you cannot have accurate tracing and observability. Each layer builds on the last to ensure that you know exactly what is happening with your applications.

MONITORING FOR THE WIN

When Bob was living his brief hacker dream lifestyle and attending meetups all the time, he went to a workshop on SQL injection. The two guys who were giving the workshop had an intentionally vulnerable app, and they kept performing injection attacks, but the alert they were expecting from the cloud provider wasn't popping up. Bob asked, "Do you have logging and monitoring turned on?" and they said no. He explained that the cloud provider "can't see" if those two things aren't turned on. They turned them on and redid the demo, and up popped the alert they were expecting. Bob felt very smart! When Bob spoke to them afterward and asked why they had turned off logging and monitoring, they said it was "too expensive." Bob gave them a talking-to about the value these services provide. He also worked to find out how they could fine-tune settings like volume and retention so they could cut costs in the future.

Blocking/Shielding

It is possible to put a layer of abstraction on top of your applications so that all the requests and responses go through before (requests) or after (responses) your app, with the idea that this additional layer adds protection, acting like a shield for your systems. These layers can block attacks or other potential issues for your apps. Examples of this are web application firewalls (WAFs), runtime application self protection (RASP) systems, content delivery networks (CDNs), API gateways, service mesh, object-relational mapping (ORM) frameworks, parameterized queries (a security control within your code, perhaps not technically a layer of abstraction but still a shield), or even something homemade. Look at Figure 14.1 to see an architecture with several layers of protective abstraction or "shielding."

NOTE *Virtual patching* means adding a regular expression to your network edge (for instance, to a WAF, RASP, or CDN) that matches an attack pattern you are concerned about, to block such attacks. If a vulnerability, such as Log4j, is being exploited in the wild on many systems around the world, and your organization is concerned, applying a virtual patch gives you extra time to respond, upgrade, patch, or update your systems. It can be a real lifesaver.

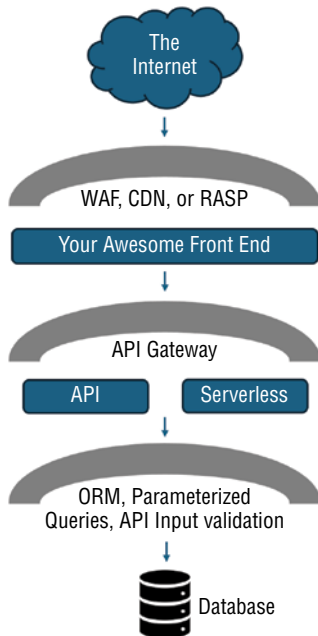


Figure 14.1: Potential layers of abstraction

Web Application Firewalls (WAFs)

A WAF creates a shield between a web app and the internet; it can help mitigate many common attacks.²⁹⁵ Usually, WAFs are installed on a load balancer or another endpoint (other than the web server). They cause a small amount of latency, and they evaluate responses to and from the web app. WAFs can be set to alerting or blocking mode, telling you if they think they see something wrong or stopping suspicious requests and responses, respectively. You can set rules on a WAF to be more or less strict, but as you make a WAF stricter, there is a potential for blocking legitimate business requests, which means you need to balance both sides of that equation. They can not only block bad requests to your app but also stop bad responses (for instance, if you were about to return the entire contents of your database, the WAF might decide that it seemed like an illegitimate response, and it would be blocked). They also make virtual patching possible.

Content Delivery Networks (CDNs)

A CDN is a geographically distributed group of servers that caches content close to end users to ensure very fast loading of internet content, such as HTML

pages, JavaScript files, images, and videos.²⁹⁶ CDNs can also help filter/block malicious traffic, mitigate denial of service (DoS) attacks, and protect against other cybersecurity threats by serving as a secure proxy between users and the server.²⁹⁷ Although security is not their first concern, CDNs offer an additional layer of defense. They also sometimes offer virtual patching, although usually only for high-priority vulnerabilities actively being exploited in the wild.

Runtime Application Self-Protection (RASP)

RASP is an application security technology that uses runtime instrumentation to detect and block computer attacks by taking advantage of information from *inside* the running software.²⁹⁸ RASP is installed inside an application, a process that is called *instrumentation*, and thus, has a very different context than most other shield-type systems. These systems cause a small amount of latency, protect the application from the inside out, and are technology dependent (they must be compatible with your chosen language and framework). Installing these types of systems is complex and sometimes quite time consuming, but they also seem to provide high-quality results. They also make virtual patching possible.

Virtual Patching

Virtual patching (also known as *vulnerability shielding*) is a safety measure against known vulnerabilities being exploited in the wild. Virtual patching works by implementing layers of security policies and rules that prevent and intercept an exploit from taking network paths to and from a vulnerability. More plainly put: a virtual patch is one or more regular expressions (REGEX) used as a block list that the input to your app must “pass.” It blocks potential exploits by disallowing anything that looks like the attack code. We perform virtual patching if there is high risk and either no official mitigation is available from the vendor or the mitigation available will take an extended period of time to apply. A virtual patch can give you days, weeks, months, or even years if it’s good enough and the risk is high enough.

API Gateways

An API gateway is a piece of software or a cloud vendor-provided service for publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at scale.²⁹⁹ It provides rate limiting, throttling, resource quotas, authentication, authorization, and sometimes even input validation. Personally, I would never put an API on the internet unless it was behind a gateway; they help stop bots and brute-force attacks, among other helpful features. They assist in making complex architectures easier to manage and more secure. Many API

gateways can be used with serverless apps as well, for an extra added layer of defense for your applications.

NOTE Although we've already discussed ORM frameworks for databases and parameterized queries, I wanted to mention that they are both examples of layers of abstraction that provide protection against threats to software and data.

Every additional layer adds more protection, which is an application of the principal of *defense in depth*, as we learned earlier in this book. Although none of these layers of abstraction is considered perfect, they each reduce risk. They also add complexity, time, and cost. You should work with your application security team to figure out which ones are worth doing for each system. Note: input validation and stored procedures are not optional; they are always worth it!

A Special Note for Data Scientists

If you and your team need to query your production database or data lake regularly, you can put an API and an API gateway in front of the database with input validation, an ORM, and stored procedures. This will add access control, help you apply least privilege, add logging and auditability, prevent injection, etc. You might think adding all of that in front of your database is a pain, and you would be right. But I assure you that having a giant security incident, accidentally polluting your valuable data, or a public data leak would be much, much more painful. I've trained data science shops before and have found that they often do not follow any software development best practices. I urge you to take all the lessons in this book to heart, especially this section about layers of abstraction and shielding, which would be an excellent place to start if you and your team are currently querying production data with no safety net.

TIP Although there are many articles about how to bypass the protection of a web application firewall, stored procedures, etc., they still greatly reduce the number of attacks your application will face by eliminating them before they get to your app, database, or API. They do make you safer, even if none of them are perfect. That said, you should *never* use a shield or layer of abstraction as a replacement for secure coding; ensuring that your app is safe, rugged, and reliable is always priority number one. Everything else, including shielding, is just gravy.

CDN TO THE RESCUE!

Alice was working at a company during the dreaded Log4j outbreak of attacks during Christmas 2021. Luckily for her, the company's apps were behind a CDN that took action very quickly and deployed a virtual patch for all customers. Alice was very

relieved! She heard her friends at other companies panicking and not being sure what to do, and her company's CDN had just handled it for them. Alice made a note to renew their contract and then went on Christmas vacation!

Continuous Testing

After we release an application, we continue to be responsible for it. We must continue to ensure that it is safe to use, as well as reliable and robust. To do this, we must continue to test it on a regular basis because we know that software does not age with great poise.

When I say *continuous testing*, I don't mean that software is being tested every moment of every day; I don't feel that's necessary. But I do mean that it needs to happen significantly more often than once a year. Exactly how often will depend on the risk your organization is willing to accept for that system. If the system is mission critical, is facing several serious threats that you are aware of, and contains extremely sensitive information, you will want to test it very often. However, if the system is low priority, without sensitive data, and there are no big threats to it that you are aware of, less often is generally considered acceptable.

Testing does not need to mean a penetration test. Testing can be scanning your software with a software supply chain security tool/software composition analysis (SCA) tool every week, doing a dynamic scan (dynamic application security testing [DAST]) once a month, also doing a static application security testing (SAST) scan once a month, and then fixing anything you find disconcerting. Then you could follow that up with a penetration test and threat model if the application had major changes (such as a major refactor or a large new feature added). You could very easily set up SAST and SCA tools to scan your entire code repository every week. The difficult part is having someone review those results and decide what's worth fixing, as it can be quite time-consuming. But your AppSec team can help with that.

The idea of continuous scanning/testing is that you always understand the security posture of your systems. No one can afford to have a penetration test done on every system every week. And quite frankly, I think that would be a waste of money. But you *can* have automated tools run very often, which can give you a good understanding of where you are at any given time.

I suggest, at the very least, getting weekly SAST and SCA scans running against your code repository. There is no risk of damage with these types of scans, and by looking at your applications from these two angles, you can find quite a few types of security issues and then fix them.

INTEROPERABILITY WIN

Everyone knows that Bob *loves* automation and smart devices. Bob has his coffee start on its own every morning at home, he has automated locks on his house with a video camera so he knows “who’s come a-knocking,” and he automates every single thing he can at work. Whenever Bob purchases a developer or security tool, he always checks to see if it works with the company’s IDE, continuous integration and continuous deployment (CI/CD), bug tracker, and code repository, at the very minimum. If it’s not compatible, generally Bob won’t buy it. If you can’t automate it, is it really worth it? That’s Bob’s motto.

Security Incidents

An IT *security incident* is any attempted or actual unauthorized access, use, disclosure, modification, or destruction of digital information or information systems. This includes interference with information technology operations and violation of your organization’s policies.³⁰⁰ Examples of security incidents include data breaches, an attacker accessing someone else’s accounts or data that they should not have access to, or intentionally making a system inaccessible (denial of service). This could also be as simple as an employee deleting an important file or inadvertently sharing confidential data. The act doesn’t have to be intentionally malicious to be a security incident.

During a security incident, the incident response (IR) team may ask for your help. If they ask for your help, usually it’s because you are the subject matter expert (SME) on the system that has been attacked, but it could also be to ask you to write a script, access some data for them, or give them background information.

During a security incident, the IR team will need the following things from you, at a minimum:

- Please treat whatever they ask you for as a top priority. This is an emergency; please treat it as such. If your boss wants you to work on something else, tell them to speak to the incident manager, and they can sort it out.
- Follow “need to know.” Do not discuss anything about the incident outside of the IR investigation team unless you have been given permission to do so. If your boss wants details, tell them to speak to the incident manager. Unless someone truly *needs* to know to do their job, you should not tell them. Let the incident manager decide who does and does not need to know. Someone *wanting* to know is not good enough, even if they are your boss.

- If you see something that looks suspicious, tell the security team. This applies all the time, not just during an incident. If you see something, please say something, even if you are not sure. The security team will not be upset about a false alarm, but they will be incredibly grateful if you've found something scary.
- If you are part of the incident investigation, tell them if you need to leave or sign off for the night. Give them a phone number where they can reach you. Don't just disappear; they might still need you.
- Do not try to "be a hero" and handle an incident by yourself. Always report it to the security team and follow their lead. They have training and experience so that they can do the absolute best job in this situation. They know how to handle evidence so that it is not spoiled and unusable in court (chain of custody). They have also agreed to certain risks when they signed up for the job, which you did not. An untrained person handling an incident can make it much, much worse than it needs to be; please do not do this.

If you're lucky, you will never be called on to be a part of a security incident, and your company will never have a major breach. That said, if it does, and if you follow these rules, you will be a helpful resource to the IR team when they need you the most!

CANADA'S LOVE OF FIGURE SKATING

Alice was becoming friends with Faythe from the Governance, Risk, and Compliance team at work and, when they had a few drinks one night, found out that she used to be an incident responder. Incident responders are not supposed to tell stories, but Faythe just had to tell Alice about this one. She responded to a security incident many years ago where a helpdesk person had tried to handle the security incident himself. She got called into the big board room and found all the executives at this large company panicking and saying that one of their buildings had malware. Faythe knew that the building in question was not internet connected or "smart," and therefore it could not have malware. The helpdesk guy had told all the executives that there was no internet coming from that building because it had malware, they were in danger, and he "knew what he was talking about." They all believed him, despite it making no sense. Faythe's team turned on Wireshark and in about 10 minutes figured out that everyone was going to the same site: a live stream of Canada competing for a gold medal in the Olympics. Alice said, "Well, everyone knows how much Canadians love figuring skating." Faythe blocked the site, put the event up on a big screen in a meeting room for everyone to watch, and then told everyone to "stop streaming the damn Olympics and clogging up the internet pipe." It took about 20 minutes to fix everything. That helpdesk guy, though . . . Faythe ended up spending weeks explaining to the entire company over and over that "No, there was never any malware," and "Yes, she was sure," and "Of course, Canada should have gotten gold;" she couldn't believe they got silver, either. Alice and Faythe were in a fit of giggles by the end of the story.

Business Continuity and Disaster Recovery Planning

Business continuity is the capability of an organization to continue the delivery of products or services at pre-defined acceptable levels following a disruptive incident. Business continuity planning (BCP) means creating systems of prevention and recovery to deal with potential threats to a company and then documenting it. In addition to prevention, the goal is to enable ongoing operations before and during execution of disaster recovery. Business continuity is the intended outcome of proper execution of both business continuity planning and disaster recovery.

— *Wikipedia*³⁰¹

Tanya's definition: A BCP helps organizations return to normal business functions after a disaster has occurred. The BCP covers the entire business, including people, where they work, and the tools they need to get their jobs done. For example, the BCP could cover how to quickly get new computers to your staff if several machines were ruined by a power surge, how to set up a temporary work-from-home program, or an alternative way to collect money from your customers if your credit card processing system is down for an extended period.

Disaster recovery (DR) is an organization's ability to restore access and functionality to IT infrastructure after a disaster event, whether natural or caused by human action (including human error or malicious activities). DR is considered a subset of business continuity, explicitly focusing on ensuring that the IT systems that support critical business functions are operational as soon as possible after a disruptive event occurs.

— *Google*³⁰²

Tanya's definition: A DR plan details how to get your IT systems back up and running (even if only at the bare minimum) after a disaster has occurred. An example DR plan could cover how people will work (digitally) if the usual place of work (and therefore IT network) is no longer available due to flooding or how they will publish their code if their cloud provider goes out of business.

Your applications need to be part of the DR plan and BCP. The systems you create are extremely valuable and are (usually) a central part of the way your business runs. The business depends on them to provide services and products and even to run the business itself. Without them, most organizations cannot function properly, and thus, your applications need to be part of these two plans in case of emergency.

Not only do you need to have the applications themselves as part of the plans and accounted for, but the systems you use to build and maintain your apps need to be accounted for as well. Your CI/CD servers, your code repository, any sandboxes or testing environments, and everything you need to do your job must be part of the recovery plan so that you can get back to business as usual, as fast as possible.

TIP **Tips for DR and BCP:** Think of each tool you use during a regular work week and what you could use instead if it were no longer available. For instance, perhaps you use Slack or Teams to chat with your colleagues. It would be a great idea for you to all install Signal or some other app with end-to-end encryption and then exchange contact info so that you would be able to still contact everyone you need to if your regular messaging app of choice were no longer available. Perhaps you could plan to save all deployment artifacts in a secondary location (a secondary cloud provider or at a secondary on-premises location) so that you could redeploy what is in production at a moment's notice if you needed to. Plan for a different way to be able to do all of the most important functions your team performs, and you will have a good plan.

If you are a senior software developer, manager, or even team leader, you may want to ask about where your team fits into the BCP and DR plan. You will want to ensure that you have your bases covered so your team can get up and running after a disaster.

Examples of a disaster could be your building being hit by lightning or a flood, a power outage, ransomware, your cloud provider announcing it's had a fire or flood, and it has to turn off all of its machines for an entire week (including yours!), an epidemic, or a war. Anything that could interrupt or damage the fundamentals of the way your business is run would mean you would need to enact your BCP and DR plan.

TIP During the start of the COVID-19 epidemic in 2020, you may have noticed that some companies didn't skip a beat, whereas others took a long time to get back up to speed. The ones that did well were those that had great business continuity planning. Hats off to all of you who planned well.

LIGHTNING STRIKES TWICE

Bob listened to a podcast about how one of the Azure data centers was hit by lightning, caught on fire, had a flood, and then got hit by lightning again, all on the same day. It was a gigantic storm. Apparently no one was hurt, and the podcast guest detailed how they got all the traffic routed to other data centers almost immediately and then got the damaged data center back up and running in under a week. Bob was astounded! He had always thought Microsoft was rather impressive from the outside,

but this felt like a miracle. It turns out Microsoft takes its DR and BCP planning very seriously, and it doesn't just plan; it practices and tests its plans. It's a good thing it did, in this case! Bob went to work that day all riled up about finishing their BCP as soon as possible.

Chapter Exercises

You can find the answers to these questions in Appendix B, “Answer Keys.”

1. Why would you want your software to be a part of disaster recovery (DR)? What's the risk of not being a part of it?
2. What's the difference between disaster recovery and business continuity?
3. Explain what “need to know” means in your own words. Also explain why it's important when responding to security incidents.
4. Give one or more reasons why we should continue to test our software after it's been deployed.
5. Why would we want to have one or more layers of abstraction or protection for our applications? What would be the purpose of having more than one?
6. Do data scientists also need to follow secure coding best practices? If so, why? If not, why not?

Conclusion

Although this is the conclusion, we are going to continue to learn new things throughout this entire chapter. I want to set you up with a few final thoughts, habits, and ideas to nudge you in the direction I have been trying to propel you this entire book toward being a security-focused software developer. You don't need to switch to the security team (although they could certainly use your help) to make a huge, positive difference in your organization. Just loaning this book to a colleague when you're done with it can change someone's whole (programming) world. Let's jump into your secure coding future together.

Good Habits

In my first book, *Alice and Bob Learn Application Security*, I shared several good habits I thought any technologist would want to make a part of their life. In this section, I will summarize them and add a few more that apply specifically to developers:

- Every time you search online for how to do something in your code, look for the most secure way to do whatever it is. It will take slightly longer, but your code will be much more secure. Whatever “solution” is voted to the top of most online forums is generally the least secure way to solve

the problem. Not copying and pasting upvoted-but-also-wildly-insecure code into your apps will improve your code quality immensely.

- Turn on multifactor authentication (MFA) on all your accounts that matter to you. Banking, governmental accounts, health-related accounts, your email, accounts with saved credit card numbers, and anything for work—turn it on. It's the #1 thing you can do as an end user to protect yourself against credential stuffing and data breaches. Please do this right away.
- Get a password manager, and then use it to reset all your passwords on important accounts to random, unique values generated by your password manager. Then use it for your MFA authenticator as well, for added ease of use. Finally, install the password manager's plugin into all your browsers. Set up an automated backup for your password vault as well; it will be very valuable once you're done filling it up.
 - Note: You may not want to install the password manager onto your phone if you store your MFA challenges in it, as if someone compromises your phone, they have it all. You may prefer to store only important passwords on your phone's password manager instance (one vault of passwords you need often instead of all of them).
 - You might choose to have a second, different password manager for your MFA challenges, with a different password to get to them.
 - The previous two items are additional security precautions. Personally, I do not do either one. You make your own choices.
- Register for [HaveIBeenPwned.com](https://haveibeenpwned.com) with all your email addresses (unless your password manager offers this functionality to you already). It will notify you if you are in a data breach that it is aware of. The notifications are sometimes late (it can only tell you once it is aware, and sometimes it takes years for HaveIBeenPwned to be notified), and sometimes there is no notification (not every data breach is shared). That said, it's an extremely helpful service. If it sends you a notification, rotate that password immediately and check your account to see if there has been any unauthorized access. Hat tip to Troy Hunt, Charlotte, and the rest of the HaveIBeenPwned team.
- Use a passphrase or passkey (more in the next bullet) to access your password manager. A passphrase is literally a sentence that is only known to you. Please do not use a very well-known sentence, such as a famous song lyric or a famous quote. The one I use for my password manager is an entire joke (question and response). It's very long, but I can type it quickly now, and I will never forget it. It would take a very, very, very long time for a computer to crack it because of its length. Do not use something simple for entry into your password manager; use something as long as you can tolerate. It's literally the key to your online kingdom.

- Consider using passkeys. They are passwordless and easy to use. You can save them into your password manager. Anything you can do to make security easier for you is a win.
- Don't reuse passwords. It's easy, and criminals know people do it. Teach the people in your personal life about password managers and why password reuse is a very bad idea. You might save someone you love from a terrible fate.
- Get security tools and run them against your code, even if the security team doesn't supply them. Hopefully, your security team is awesome and provides you with support, tools, licensing, etc. But if they don't, please consider taking matters into your own hands. There are lots of free tools you can use that will help you write better code (and pass your next pentest).
- Consider joining security communities such as OWASP, "Let's Talk Software Security," and other free communities that welcome software developers *and* security folks into the fold. Not only will you learn a lot, but you will also meet some fantastic people. I've met a lot of my closest friends via various communities and cannot recommend joining enough.
- Avoid technical debt whenever you can. Technical debt is almost always also security debt, and that's a losing game. Do your best to stay on top of updates, upgrades, dependency management, etc. I know it's not sexy, but it is a truly important and valuable activity.
- Ensure that your team keeps an accurate and up-to-date inventory of the web assets you are responsible for. Ideally you would have a list for everything in your entire org, but you can likely only control your own team. List where someone can find the code, find the running apps, different environments where the databases are, contact information, etc.—anything you can think of that might be useful. This document will be helpful not only for the developers where you work but also for the security team.
- Some obvious advice for you: don't leave your machine unlocked when you walk away, don't download random software or images from the internet and install them onto your work computer, don't visit questionable websites with warnings on them, and don't use your work computer for nefarious means. I suspect you already knew all that, but I would be amiss not to mention it.
- Be aware of what private information you share about yourself online. Don't post pictures of your work pass, your date of birth, your home address, etc. Also, don't share digital images of you that are compromising unless you are at least somewhat prepared to see them on the internet someday. Please be extra cautious about the images and videos you share of yourself on dating apps or with people you are just beginning to get

to know. If it is something that could be used against you someday, perhaps to blackmail you, please think twice (or several times) before sharing it with someone you do not have complete faith in (including their technical ability to protect said data).

- Be aware that you are a superuser, and thus, you are a target of malicious actors. Software developers are *powerful*. You make something out of nothing all day long. Someone with a grudge against your company would love to have access to your accounts, so please assume a strict stance when it comes to protecting your work devices, accounts, data, and everything else.
- If you aren't sure, ask the security team. It's okay to ask for help, and it's literally their job to help you do your job securely. As a person who is really bad at asking for help, I am telling you to do it because I know many of us think, "I can handle this on my own," or "I don't want to bother them." Please don't think like that in this case. They are experts who are hired specifically for this purpose, to help you; make them earn their paychecks.
- Be suspicious of anything that seems too good to be true. This could be emails, support that someone offers for your open-source library, or anything else. If it comes out of the blue, try to validate whatever it is from a secondary source.

BOB LOVES HIS PASSWORD MANAGER

Bob used to practice "password reuse" but learned the hard way why no one should do that and started using a password manager years ago. Surprisingly, it's saved his bacon a few times, from telling him when his credentials had been in a breach to saving his credit card information and even warning him when his card was about to expire. One time, he clicked a phishing link: he was so tired that day, and it looked *just* like his bank's email. When he went to the website to log in, his password manager wouldn't work. It refused to help him log in. Then he looked very closely at the address bar and realized the website of his bank was spelled wrong! How terrifying! He almost gave away his banking password! Bob's love for his password manager grew.

Your Responsibility

It is your responsibility to do your job as safely and securely as you know how, as well as to follow any policies, standards, and guidelines your organization has provided. This means writing and designing apps as securely

as you know how to, fixing the bugs you are asked to fix, and using the tools provided to you. Everything above that is a bonus. This means if you decide on your own time to take a secure coding course, read this book, and study threat modeling, that's above and beyond your responsibility (it also makes you *amazing*).

It is also your responsibility to notify your boss and/or the security team if you see something disconcerting. This could mean telling them that your team is considering designing their own encryption/authentication/authorization/identity/session management/some-other-security-control rather than using the approved module/approach/product. It could mean reporting that your API is exhibiting extremely strange behavior that your team cannot explain. If there are security bugs you are aware of but that aren't being tracked anywhere, ensuring that they are documented would be a part of this responsibility. Perhaps you saw something on the internet about your company, and it has you concerned. Tell them. Trust me, they want to know.

The weight of the entire security program is on the security team, not you. Your job is to take care of your part: the software you write and maintain. After reading this book, you should be in a very good position to do an excellent job from now on.

DON'T KEEP THE WRONG SECRETS

Last year, Alice hired a penetration tester, and afterward, one of her developers told her the tester had been really good. Alice had already known that; she had checked his references, and he came highly recommended. But she was curious what the dev meant. She said, "Oh yeah? How so?" and then let the dev speak. He told her that he had a list of vulnerabilities he knew about, and the previous two testers had missed them, but this guy found all of them, and, therefore, he was a great tester. Alice was outraged! The developer had known about multiple vulnerabilities and not reported them!?!? She needed to think about what to do with this information and not make a rash decision in anger.

How Much Is *Enough*?

Something I struggled with a lot at the start of my cybersecurity career was "How do I know that I have done *enough*?" I worried incessantly that I had not brought my organization up to a security posture that was appropriate. As I moved forward in my career, I realized that some of the places I worked didn't have a specific level of assurance they demanded; most of them were in a panicked state, as though they were drowning, trying to put out one fire at a time. I did not enjoy that feeling. I always felt like I was not doing a good enough job and that my work was never finished.

Once I had more authority and experience, I started setting specific goals. We would look at the entire program and decide where we felt we needed the most work and what level of security was “good enough” for us, and then I drove us there as fast as I possibly could. Having measurable and achievable goals means I can go home at night and feel my work is complete (for that day). That’s something that matters to me and helps me feel a lot less stress about my job.

If you work at a place where there are no specific goals, standards, or service-level agreements (SLAs) and it seems that there are no specific expectations from you regarding security, ask. Besides the obvious of fixing security bugs that are assigned to you and not breaking policy, I suspect the security team does have things they wish you would do, whether they have communicated that clearly to you or not. Asking them what security steps are supposed to be part of the system development life cycle (SDLC), if they need your help for anything, if they plan to offer secure coding training, etc., is very reasonable to inquire about. If they say they’ve got nothing for you, then you’re all set. However, that may not prove very satisfying to you.

That said, I suspect that they will have all sorts of things they would like to do with you. They may invite you to become a security champion or participate in a threat model, or ask you to review ideas they have and provide feedback to help make them more developer friendly. You might become their new best friend forever (BFF), a developer they can rely on.

S.M.A.R.T. GOALS

Setting goals in your personal and professional life is extraordinarily important. I’m often shocked to find out how many organizations I work with have no goals for their AppSec programs and people who have no career goals other than “don’t get fired.” When you set specific goals that are measurable and are very clear when you have met them, you are more likely to reach them. If your goal is to “put on 5 pounds of muscle in the next 6 months,” “speak French well enough to travel throughout France for 2 weeks and not need to speak English,” or “have no high vulnerabilities found on my team’s next penetration test,” you know if you’ve met your goal or not. You need to plan out the steps you will need to get there and the time to do each step. For instance, lift weights every Tuesday and Thursday and have three protein shakes per week, take a conversational French course in night school for 6 months, or run a static application security testing (SAST) and software composition analysis (SCA) on your code and fix everything they find before the pentester comes in. I realize I’ve simplified the previous goals, but I hope you get the picture. The more detailed and specific you can be, and the more people you talk to about it (to keep you accountable), the more likely you are to *do the thing*. If you want to accomplish something *big*, you need to be deliberate and intentional about it. No one casually writes a book, starts a new business, or learns a second language; you need to work your butt off. I want *you* to accomplish huge things. Consider setting one or more personal goals that matter to you. Trust me, it feels amazing when you’ve pulled off something you’ve always dreamed of.

NOTE Please don't overwhelm yourself by choosing too many professional goals. Remember that free time, happiness, self care, and spending time with people you love should also be part of your goals and everyday life!

Using Artificial Intelligence *Safely*

Ever since ChatGPT went public, I have seen many organizations using artificial intelligence in unsafe ways. They give it too much control and allow it to make decisions without oversight. They do not validate the results they receive from it before using them. They give it sensitive data that they should not be sharing. They do not test it thoroughly enough. They don't perform threat modeling, and predictable problems happen over and over again.

There are books that dive deeply into this topic, such as *Uncontrollable* by my friend Darren McKee, which I suggest you read if you have concerns about AI safety in general. This section is going to cover defending against high-level concerns as they relate to software development:

- Do not allow the AI to make decisions on behalf of your applications without oversight. Each decision the AI makes should have some other part of your code (not controlled by that same AI) validate the decision. An AI should never be trusted to make an important decision on its own. An example would be to give a refund to a customer, to change the date of a plane ticket, or to delete an account of a misbehaving user. If the AI is deciding to do these things, you must have a second process that validates the decision before it is actioned.
- Never allow the AI to have agency over your systems, organization, or data. An AI should not be able to control itself or others. Right now, researchers openly admit that they do not fully understand how AI models work, which is why they are unable to explain why they continue to have hallucinations. With us not being able to understand as an industry exactly how they work or why they work, we certainly should not be allowing them to have control. Whenever you watch a sci-fi movie about AI taking over the world, it's always because they've been given too much control. I'm not trying to scare you; I'm trying to avoid obvious and stupid mistakes. Allowing a system that we do not fully understand to be able to make major decisions on our behalf is obviously going to result in stupid mistakes. We've already seen it repeatedly.
- Do not share sensitive or private information with an AI unless you own/control it and your security team has approved you to do so. I fell into this trap already myself. I was using an AI to help me write marketing copy when I was preparing for my company to be acquired, and I told it

about the acquisition. We were announcing the business deal only two or three days from then, but it was still technically sensitive and confidential information. What if the AI had given that information in answers to someone else and then they found out about it ahead of when our announcement came out? This is a small example involving barely sensitive information. Several people had already publicly guessed on social media that Semgrep was about to buy We Hack Purple. But imagine if it was something more serious? Or a much larger deal? What if you were asking the AI about very sensitive health information? The AI knows who you are when you use it, and in questioning it, you are essentially telling it, “I’m really concerned about health condition XYZ.” It also likely means that probably you or someone very close to you has that health condition (unless you are a medical student or professional). Do you want to tell the world about your health condition? No? Perhaps think twice before sharing that information with the AI. This goes for information at work, too!

- If you are asking the AI to write code for you, you need to review that code before you put it into production. You need to review that code before you save it into your code repository. You should also run a SAST tool and an SCA tool on it to have it double-check that there’s nothing you missed. Just like you wouldn’t take a random image from Docker Hub and throw it into production without testing and validating that it’s safe to use first, you shouldn’t copy code out of an AI and throw it right into production, either. I know it’s easier that way, but it’s always easy to do a bad job. You want to do a great job, and that means reviewing the code before you use it. If you do not understand how it works, you can ask a friend or colleague. But unless you understand the code it gave you, I recommend that you don’t use it.
- Try to make sure that the AI has not broken copyright in what it gives you. It’s surprisingly common, and being sued is never fun for anyone. You can check by taking what it gave you, putting it in quotes, and searching for it online, for starters. If it’s an image, you can do an image search. You will also want to ensure that you have the right to use what the AI created for your work (that it does not hold the copyright, as opposed to you, to what it has created for you). Consult your legal team if you need more information.
- Always double-check every single thing the AI produces. Even if you’re tired and it did it right the previous five times. Trust me on this.

How you should use AI to your advantage:

- Use the AI to write user stories, documentation, and anything else that is written for your job. Then double-check everything before showing it to anyone else. Remember, the first draft will likely be . . . imperfect.

- Use the AI to help you write code; just make sure you check it first and don't give the AI sensitive data when asking for that help.
- Use the AI to help you find vulnerabilities! If you are allowed to share your code (for instance, if it's open source), feed it into the AI and ask it to find vulnerabilities. It might not be as good as a SAST tool, but if you have an AI and you don't have a SAST tool, take what you can get!
- Ask the AI to do threat models for you. It will likely come up with lots of ideas you find ridiculous, but I find it usually has one or two good ones.
- Ask for help with design, describe what you want, and see what it comes up with. I often find that asking the AI is sort of like asking a junior employee: the ideas can be a bit wacky, but you get a lot of results really fast, and some of it is usable.
- Ask the AI to add comments to your code for you. It might be able to add more than you would have the patience for. That said, please be concise and brief. No one wants to read a novel while coding. It's a careful balance.
- Ask the AI to suggest bug fixes. Some of them will be good!
- Be creative; I'm always surprised by the fun stuff I can get an AI to do.

Continuous Learning

When you decide to work in technology, you agree to be a lifelong learner. You cannot excel in this field if you do not learn new things. What follows are some strategies that have worked for me; please adopt any that work for you!

- If it is your job to write code, I suggest you learn secure coding.
- If it is your job to design software or you are an enterprise architect, learn secure design and threat modeling.
- I suggest you figure out how you learn best. Every single person is different, and it only really matters how you learn best, not anyone else. I know that I like to read or see something, try it out for myself, and then show it to someone else. That's how I started teaching and doing workshops: teaching it to someone else or writing about it for someone else cements the knowledge in my brain and helps me learn it for life. Try to find the learning style(s) that work best for you, and then use them.
- Try finding the format(s) of learning that works best for you. Although I really enjoy a live talk or training, that's an expensive way for me to obtain knowledge, and I've learned that on-demand online courses work pretty well for me, too. I like on-demand because then I can watch the video a

few times if I need to, especially if I space out and start thinking about other things (which is extremely common, and therefore you should account for that if you know it happens to you). That said, I find a live trainer a lot easier to focus on, especially if the teacher is really good. You may find that reading a physical book is better than listening to audio, or hands-on exercises are better than discussion. Figure out which formats work the best for you, and try to find training that includes them.

- Try to find out which are the more affordable ways for you to learn. When I first started out, I just looked for anything that was free on the internet and books from the library. Although that was free, it took a long time, and the content I consumed was very low quality. Eventually, I figured out that I could get into conferences for free if I spoke at them, and because I really like the live aspect, that worked *great* for me. I also try to make my workplace pay for learning opportunities whenever possible (including job shadowing or mentoring during work time). Other options include podcasts, newsletters, meetups, study groups, books, audiobooks, hiring a trainer to come into your office, going to preconference trainings, starting your own open-source project, and getting a part-time job. There are a *lot* of ways to learn.
- Ask about job shadowing at work. Find a professional mentor. Ask to join a project on another team for the experience.
- Tell your boss or mentor your career goals, and ask for direction. They tend to have amazing ideas of how to get there that you may not have thought of.
- Set a regular time to check in with yourself and your goals and see where you are. If possible, have an accountability partner to ensure that you stay on track.

TIP I took a training with Adam Shostack last year, and he's a great teacher. There were interactive activities, lectures, and discussions, and it was easy *for me* to learn from him. The way he taught worked *for me*. That's not always the case; not every teacher is created equal, and they all teach in different ways. If you find someone that you think is a great teacher, I suggest absorbing as much of their content as you possibly can, even if it's not 100% the topics you're looking for, because it will be easier for you to learn from them than someone else. It will also encourage you to keep learning.

Becoming a Champion

A *security champion* is the person who acts as a liaison between their team and the security team. They usually receive special training and have additional

responsibilities and additional time with the security team. It looks great on your resume, you will learn how to be a very security-conscious developer, and if you were ever considering getting into security, this is a huge step in the right direction.

Being a security champion means volunteering to lead your team in doing all the things the security team needs from you. This can mean checking all the automated tool results and ensuring that the important ones are actioned, ensuring that security-related parts of the SDLC get done and the security team is informed, that you meet with the security team regularly and give them updates, that your team meets certain SLAs, or something else. Each program is different, and, therefore, the expectations are different.

You might be thinking, “What’s in it for me?” For one, if you become a champion, you will be performing good. When I say “good,” I mean how Superman performs good; you are protecting others (your colleagues, investors, the public, and/or customers). You are also making yourself more visible within your organization, and when it comes time for promotion, the person who went above and beyond is usually the first choice. You will get to learn a lot of cool stuff, including how to write better code and design better apps and what types of risks and threats your systems face on the daily. I realize that I’m biased, but I think this stuff is fascinating. Security champion programs often contain different types of rewards, such as getting an afternoon off work, free food, or gifts such as books or challenge coins. You will also learn skills that will help you protect yourself, your loved ones, and your personal devices for the rest of your life. It’s a pretty good deal!

I strongly encourage you to volunteer to become a security champion if given the opportunity, even if it’s just for a few months. Tell them you’ve read this book, and you’ll be a shoo-in.

MY CHAMPION

From my very first AppSec program, I have run security champion programs, formally or informally, as part of every security program I have run full-time. It helps me scale my team and build trust. Quite frankly, I love people, and security champion programs mean I get to meet a lot of cool software developers, who tend to be my favorite type of person. I remember in high school thinking that of all my classes, the people I liked the best were in my programming classes, and if I was going to choose whom I got to work with every day of my career, it was them. Recently, one of the security champions from my very first program told me he just started *his* first security champion program! I knew he switched over to security a few years ago, but having him start a program for his own org feels pretty wonderful. Liam: you rock!

Getting Others on Board

When “Tanya from the security team” tells a software developer what to do, it is not at all the same as when you tell your teammate, “Hey, we’re supposed to do it like this now.” You are their peer, whereas the security team is often seen as either strangers or outsiders at best. You probably have a lot more influence on your team than you realize, and if you suggest that others follow the secure coding guideline, the new security policy, attend the threat modeling session, etc., people will probably do it. Especially if you go with them. Inviting people along with you makes it less intimidating *and* more fun. Win!

Try to make a point of mentioning it when you see someone doing something you know they should not do or if they’ve missed something you know they should have done. If you find a bug or a neat solution for something, consider discussing it with the team and documenting it if that makes sense. Talk about security. Talk about issues. Ask others their opinions. The more you bring it up, the more likely others will too. It will slowly change the culture of your team for the better.

TIP If you have found something someone did wrong, be gentle when you bring it up. Try to think about how you would want someone to break the news to you, especially if the mistake is something really bad. Tell them with respect and empathy. No one likes being told their baby is ugly; please remember this when delivering less-than-ideal feedback.

Transitioning onto the Security Team

At this point in the book, you might be thinking, “I want to work on the security team someday.” You will be in a great position to apply for a job on the security team if you have put all the things in this book into practice. As you go through this book and apply this knowledge, make sure you tell the security team what you’re up to. Mention to them that you tried doing a threat model with your team. Tell them that you have created a secure coding guideline, and ask them to review it. As you do more and more security-focused tasks as part of your job, communicating that you are doing that work to the security team shows them just how serious and interested you are.

You can also ask the security team if they need any help. Ask them if there’s a project you could work on. Ask them if there’s work they need done for your team that you could work on. You want to make sure this is okay with your boss before you start doing it, though—I do not want you getting into hot

water with your manager. Volunteering to help the security team will give you opportunities to learn, visibility, and the chance to show off your new skills.

The easiest way to transition from software development to IT security is to do it at a place where you already work. You have corporate memory, which means knowledge about all the different systems, which offers huge value to the security team. It also means there's no turnover; new employees take a long time to get up to speed, whereas you would be ready. It also means they don't have to do recruitment, so that saves money and time as well. Hiring from within, if you have the talent, is always a great deal from a management perspective. It also means you have the chance to "audition" for the security team in advance by showing them projects you've already worked on, them having the ability to speak freely with your manager, and you being able to volunteer to help with work that they need done. All of this adds up to it being the easiest way to transition into security.

TIP If you want to switch teams, tell the security team that you want to work in security. Tell people at meetups, tell your boss, tell everyone. If you keep it a secret, it's less likely to happen. Tell the whole world what you want.

Applying for Security Jobs Outside of Your Organization

Although transitioning from one job to another job at the place you already work is the easiest way to switch to security, that might not work for you. Maybe your boss won't let you switch teams and is blocking you from doing so. Maybe the security team at work already has the application security spots full, and they won't have budget to increase their headcount for the next year or two. Maybe, due to the size of the organization, there is no security team at all. Maybe you think the security team are a bunch of jerks, and you don't want to work with them specifically, but you know that security is the right move for you. Let's talk about applying for jobs outside of your organization.

One unfortunate fact that you should be aware of in the security industry is that it is quite common for job descriptions to focus only on senior positions. They will often say something is an entry-level job but expect five to seven years' experience, which is a contradiction, to say the least. As a result of this, I strongly suggest you apply even if you don't have enough experience. If it says entry-level, and you feel you're ready to enter the security industry, apply. I do suggest, however, that when you apply, you explain every single way that you have any relevant experience, even if it's not professional. For instance, make a list of all the books that you've read, including this one, to show that you are researching and learning on your own time. If you have contributed to an open-source project, if you are part of a security community, if you have

attended security conferences or courses, list all of that, too. There are quite a few online academies that offer certificates for taking courses; it's not the same as a formal certification, but if you can get a couple of those on your resume, that helps. You should also list if you were a security champion and any of the security work you did as part of your job, and write it all up in a cover letter to show just how determined you are to make this transition.

TIP I will be streaming free, live, virtual lessons for each chapter of the book starting in Spring 2025, just like I did for my first book. They will be saved to YouTube afterward in a playlist, in case you missed them. Sign up to my newsletter to be notified: <https://newsletter.shehackspurple.ca>.

When I switched from software development to security, I was pretty stubborn about it. I was at the top technical level (CS-03) at the time for the Canadian public service, I had been at that level for almost a decade, and I did not want to take a demotion when I switched over. I had already worked in software development for 17 years, had led countless successful projects, and felt that even though I was not yet a security expert I could do a very good job of anything they handed me. When I applied for a job, I performed a free penetration test and then mitigated every single vulnerability I found to prove that I could do both sides of that job. I also showed up with extremely strong references from my current and previous jobs, and I also had a professional mentor who was well known and willing to give me a strong reference that I was learning as quickly as I possibly could and that I was extremely dedicated. You should make sure you have strong references, even if they are just community members who can attest to the fact that you show up every month at the meet-up, you act in a professional manner, and you're trying very hard to learn and get into the industry.

Another way that you can transition into security is finding a professional mentor. I have had several professional mentors throughout my career, some of them formal relationships, some of them informal, but all of them helped me. They gave me advice, made introductions, and helped me avoid mistakes. I have had mentors for security, for business, for public speaking, for everything that I try to accomplish, especially if it's something new. I find someone who I think is amazing at that skill, and I adopt them. When trying to find a professional mentor, it is important to remember that they are doing an amazing favor for you for free. You really need to respect their time.

I have the following suggestions for when working with a professional mentor:

- If a professional mentor makes an appointment with you, do not miss it. Always respond to their emails. Always respond to their texts. Never, ever waste their time. Their time is a gift; treat it that way.

- Always try out the advice they give you. Unless you think it's absolutely horrific, you should try it. Even if it's a lot of work. I remember several of my professional mentors telling me they liked mentoring me because I always did whatever they asked, as best I could, and reported back. If they give you advice, you don't follow it, you don't get the results you want, and then you complain, that is extremely annoying for your mentor. If you won't follow their advice, how can they help you? If you think their advice is bad, then they aren't the right mentor for you, and it's time for you to move on. It is okay to break up with a mentor (respectfully) if it's not a good match.
- Always close the loop. If your mentor gives you advice, introduces you to someone, or provides some other opportunity for you, tell them how it went afterward. Always report back. If they ask you to read a book, even if you only end up reading half of it because you found it too difficult, explain that to them. I remember my first mentor had me read *The Shellcoder's Handbook*, and I read up to Chapter 10, did all the exercises, and decided that I was not meant to be a shell coder. I did not want to write exploits in assembler or do reverse engineering, but by reading that book, I learned that that option was not the right one for me. So that's what I reported back to my mentor. I had learned a lot, and it was helpful because I now knew that one subfield within cybersecurity was not the right path for me.
- Ask if you can help your mentor. If you can provide any help to your mentor, you should. This might mean making an introduction for them, helping one of their kids or junior employees, or even just listening to them vent. They're doing you a giant favor, and if you can pay them back in any way, you should do it. I have taken many a meeting that I otherwise would have declined, because a mentor has asked me to. I owe them!
- Acknowledge your mentors. I thank my mentors a lot. I thank them in my books, in my blogs, and when I do interviews, and I especially thank them directly. I explain my gratitude to them regularly. I also tell people that I have mentors and give them credit, often. I know that I'm the one doing the work, but having someone give me direction, make me feel supported, and give me new opportunities is invaluable to me. I want everyone, including them, to know how much I appreciate them.
- Do not expect a lot of time from your mentor. I tend to meet with one or two of mine once a month or once per quarter for 30 minutes. I have questions ready, and usually I've already left them a message about how all the things from the previous month/quarter worked out. If I can give them back some time, I do.

- Confirm with your mentor that the discussions you have are confidential. It is extremely important that you know they will keep your confidence. Stating it explicitly is worth the 30 uncomfortable seconds it takes to say it.
- Ensure that your mentor is someone you can trust and that they are not in a conflict of interest. Twice I have had mentors push me extremely hard toward a business deal where they would financially benefit; I had to break it off with both of them. It's really important that your mentor is helping you out of the kindness of their heart, not because they are going to manipulate you into doing things that they will profit from. It is absolutely okay to help your mentor, do them a favor, or work for them. But if they're in any potential for a conflict of interest in the advice they give, you might want to pause the relationship until that business situation is over or end it entirely. If your mentor gets upset when you bring up a potential conflict of interest, this is not a person you want to have as your mentor.
- It is okay to have someone you think of more as an adviser than a mentor. Maybe you only ask them specific questions about one topic, rarely, and you don't really think of them as a mentor. That's okay! Think of them as an adviser instead—a person who gives you great advice, whom you respect and appreciate. I know that quite a few people think of me as an adviser, and that works, too.
- Something else you might want to consider is having an accountability partner. This is different than a mentor or an advisor. They keep you on track and check in with you and kick your butt when you need it. If you have trouble completing projects or staying focused, an accountability partner might be the thing that takes your career or business to the next level. They can help you achieve your most aggressive goals.
- Questions you might want to ask your mentor:
 - What books or articles would you suggest I read to learn about X?
 - Is there anyone you think I should meet?
 - Are there any courses I should take or videos I should watch?
 - Are there any communities I should join or events I should attend?
 - Are there any newsletters or blogs I should subscribe to?
 - How can I go about getting experience in the area I want to learn?
 - How could I have handled X situation better? What should I do instead next time?
 - I have a meeting coming up about X. Do you have suggestions about how to communicate Y more clearly/more forcefully/more professionally/more technically/etc.?

When you have a professional mentor, you're going to need to do some introspection. The first time I had a professional mentor, I just needed to learn technical things, but very quickly I found out that I also needed to work on my soft skills. Personally, I struggle with setting boundaries and saying no, which things I should focus on, and not constantly volunteering for unpaid work because I know it needs doing and no one else is doing it. I have a bit of a bleeding heart.

Right now, I have a mentor (hi, Ilona!) who is helping me learn how to say no in a polite but firm way. Quite often, people will get so frustrated with having other people ignore their boundaries that they blow up, which is not something any of us want to do in a professional setting. I also always want to be kind, and I had to learn that you can say no but still be a kind person. I find that having someone who can be very direct with me has been invaluable. "Tanya, the problem this time is you. You said yes to too many things. Let's look at your list and remove some stuff. And then let's look at saying no instead of yes next time." At first you will likely focus on technical skills, but if you keep on learning and being mentored, you will excel in communication skills, time management, network building, business, and much more. The sky is the limit when you have an amazing person guiding you whom you look up to.

With that, let me thank a few more of the people whom I think of as my mentors, advisors, and accountability partners: Ilona, Kim, Chenxi, Daghan, Clint, Zane, Dan, Troy, and Adam. Thank you for teaching me, guiding me, and supporting me.

TIP Every Monday on X (Twitter), LinkedIn, BlueSky, and Mastadon, I run a free mentor-matching program called #CyberMentoringMonday. Please feel free to use that hashtag and tag me (@SheHacksPurple) or jump onto my weekly thread to try to find a mentor. I will retweet or repost you if I see it (sometimes X suppresses alerts, but I do my best). Explain what you want to learn, how much you've done so far to try to learn, and what your goals are (which might just be that you want to figure out which part of cybersecurity you want to work in or if you want to join the field at all). Since 2018, hundreds (if not thousands—I don't keep count) of people have found each other via this weekly thread and hashtag. Please take advantage of this if you think it will help you.

Conclusion

In this book, we covered extensive advice about how to write secure code from a language-agnostic point of view as well as specific advice for several different programming languages and frameworks. We covered how to program different types of applications more securely, such as mobile apps, Internet of

Things (IoT), web sockets, and APIs. We talked about several different types of vulnerabilities, as well as many different security activities that you can add to your SDLC to turn it into a *secure* SDLC. Along the way, there were chapter exercises, stories of Alice and Bob and their friends, and, hopefully, ideas that inspired you. All this advice should turn you into a *secure* coder and, with any luck, also a security enthusiast.

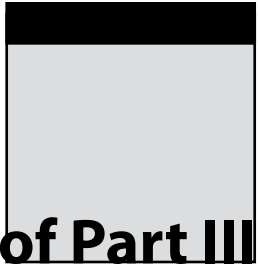
If you want to learn more, there are several options for you:

- Hire me for secure coding training, attend one of my live trainings open to the public, or take one of my online courses.
- Read my first book. Read other books on the security topics that interest you. Try to find an author you like, and read everything they've written.
- Sign up for my free newsletter and read my blog at shehackspurple.ca.
- Sign up for the weekly TL;DR Sec newsletter, which summarizes technical blogs and other learning content, for free at tldrsec.com.
- Sign up for free training online at places like Semgrep Academy (which I built), OWASP Secure Coding Dojo (which my friend Paul Ionescu built), API Sec University, etc. There are many amazing free resources online. Find one you like, and explore it.
- Watch my free videos on YouTube (youtube.com/shehackspurple). Also, watch videos from the OWASP YouTube channel; they are amazing.
- Join online communities that specialize in what interests you. For instance, I love the OWASP community and participate in their conferences and local chapters. Choose one that is right for you. Then make friends.
- Find other newsletters, influencers, YouTube channels, etc., that you like, and follow all of them.
- Find podcasts you enjoy, and subscribe. Make time to listen while you walk, drive, clean, or work out. I learn *so much* while doing laundry.
- If you find a person who makes content that you really like, consume all their content. I do this often. I find an author I like, and then I read 100% of their books. People who create content also often make it in multiple forms, such as podcasts, blogs, online courses, live conference talks, etc. If you read someone's blog and find out they make stuff in other formats, you might enjoy those, too!
- Consider getting either an accountability partner or a mentor. Or both! I cannot explain with mere words how valuable such a relationship can be.
- Make a commitment to yourself to never stop learning. Making a promise to yourself, especially in writing, will help you ensure that you prioritize learning in your career. Tell a friend, your significant other, or your mentor. Tell them about your goals, or announce them publicly on social media like I did when I tweeted that I was writing this book. Saying something out loud or writing it down makes it a lot more likely that it will happen.

After reading this book, you should feel a lot more confident that you are creating more secure applications. You should know where to find answers to your questions and where to learn more. I suspect the security team will also notice you and your new efforts. Your teammates will likely start asking you for help, and other teams will ask your opinion. People might ask to borrow this book or for you to share what you've learned with the rest of the team. You might be known as "the secure dev" and build a reputation as someone who cares about security and can be counted on for good advice. Your career might take a new, more security-focused path.

I will end this book with a call to action. I've already hinted at this over and over throughout the book; I would like to request that you share as much information as you can from this book with others. Please share any security information you think is valuable with others, whether it's from this book or any other trusted source. One of my goals for my career is to try to move our industry forward, specifically in regard to building more secure software. I want to stop reading about giant security incidents and data breaches all the time. When I fill out a form online, I want to stop thinking, "Gosh, I hope they don't lose my data," because right now, I (tragically) assume it's only a matter of time. Every single person in our industry has the responsibility to do their jobs as securely as they can, but most of them do not think that sharing information with others is part of that responsibility. It is my hope that you and I can help change that mindset. Please loan this book to a colleague or friend, share ideas you think are interesting, and adopt whatever you feel is valuable into the way you do your job. Please help me on my mission by sharing knowledge with others whenever you have the chance.

Thank you.



Summary of Part III

In this section, we covered a secure system development life cycle (SDLC): security activities you can add to your SDLC to ensure you reliably release secure software.

Checklist of Security Activities for Each Phase of the SDLC

No matter which SDLC you are following, you will need to gather requirements, create a design, write the code, test it, release it, and then maintain it. Although I have sectioned these off into waterfall-sounding naming conventions, we all must perform these tasks to create software, no matter the methodology we choose. Please try to add one or more of these security activities to each phase of your SDLC.

NOTE For each phase of the SDLC, fix the issues you find that cause organizational risk.

Requirements phase:

- Create security requirements and add them to your project.
- Invite a security representative to your project kick-off meeting.
- Ensure that your security activities of choice are in the project schedule.

Design phase:

- Perform threat modeling on your app (evil brainstorming).
- Apply secure design patterns and concepts whenever possible.
- Create security user stories if your project has user stories.
- Perform an architecture security review or a whiteboarding session with a security representative to spot potential design flaws.
- Examine all of your data flows for potential security problems.

Coding phase:

- Install security-related IDE plugins, and then use them.
- Scan your code with a static analysis (SAST) tool, a software composition analysis (SCA) tool, and a secret scanner.
- Perform manual code review, either yourself or peer review.
- When writing unit tests, consider creating unit tests to validate your security functions and also to test the security of your non-security functions.
- Once your app runs, scan it with a dynamic scanner if you have permission from your security team. If not, ask them to scan it for you.

Testing phase:

- Ask for as many types of tests as possible, and if you can choose, select these in order of priority: penetration testing, automated testing, IAST.
- If you were not allowed or did not have access to SAST, SCA, and secret scanning in the previous phase, ask for those now.
- If you run tests as early as possible, it gives you more time to fix what you find. If you can run automated tools during the code phase, rather than waiting until the testing phase, that's better; try to do testing as early as possible.
- Ideally, you are fixing issues and sending them to be retested repeatedly during this phase.

Release/deployment phase:

- If you use a CI/CD, some security tests/tools should be added, but be careful not to add slow tools or tools with a lot of false positives.

- Ensure that you lock down the release process and tooling, especially access control to said systems, and back them all up very carefully.
- Investing time and exerting effort toward making every security approval faster, more efficient, and (if possible) self-service is a win for everyone involved.

Maintenance phase:

- All systems, no matter how small, require logging, monitoring, and alerting.
- Adding blocking and/or shielding to public-facing apps will make your life easier.
- Applications in production still require ongoing testing, even if they appear unchanging.
- Not only the applications we build and maintain but also the systems we use to do that work must be part of the business continuity and disaster recovery plans.
- We may be called on to help with a security incident investigation; if so, follow “need to know” and all other instructions the incident response team gives you very carefully.
- If you see something weird that you cannot explain or something that looks like a potential cyber-security issue, call the security team, and report it immediately.

Thank you for reading this book and fighting the good fight with me and many others who care about digital security. I hope to see you on the book’s live streams, my YouTube channel, my socials, my classes, or at community events worldwide. I’m very friendly; if you see me at an event, please say hello—I would love to meet you.

I appreciate your dedication to making the world a safer place.

Tanya Janca
SheHacksPurple

Chapter 1: Introductory Security Fundamentals

1. Trusting species: <https://hbr.org/2009/06/rethinking-trust>
2. Eves: https://en.wikipedia.org/wiki/Alice_and_Bob
3. Definitions sidebar: <https://languages.oup.com/google-dictionary-en>
4. Rogers outage: <https://www.cp24.com/news/rogers-service-starting-to-recover-after-canada-wide-outage-1.5979474>
5. Least privilege: <https://www.cyberark.com/what-is/least-privilege>
6. Telegram is not end-to-end encryption by default: <https://www.trustedreviews.com/news/is-telegram-safe-4130553>
7. The various cyber security regulations: https://en.wikipedia.org/wiki/Cyber-security_regulation
8. List of several compliance options: <https://www.shiftright.io/blog/getting-to-know-compliance-in-software-development>
9. <https://nordlayer.com/blog/cybersecurity-compliance-everything-you-need-to-know>

10. The Security Knowledge Framework: <https://owasp.org/www-project-security-knowledge-framework>
11. OWASP SAMM: <https://owasp.org/www-project-samm>
12. NIST 800-53: <https://cyberexperts.com/application-security-framework>
13. <https://www.cisecurity.org/controls>
14. ISO 27001 and 27002: <https://www.bitsight.com/blog/7-cybersecurity-frameworks-to-reduce-cyber-risk>
15. SOC 2: <https://www.bitsight.com/blog/7-cybersecurity-frameworks-to-reduce-cyber-risk>

Chapter 2: Beginning

16. Input validation code example: <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/isdate-function>
17. C# syntax: <https://www.tutorialsteacher.com/csharp/csharp-datetime>
18. Minus years from a date syntax C#: <https://stackoverflow.com/questions/2992451/how-to-subtract-a-year-from-the-datetime>
19. TryParse syntax: <https://www.inforbiro.com/blog/c-check-if-string-is-datetime>
20. W3schools PHP input validation: https://www.w3schools.com/php/php_form_validation.asp
21. XSS prevention cheat sheet: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
22. .NET and PHP HTML encoders: <https://www.web2generators.com/html-based-tools/online-html-entities-encoder-and-decoder>
23. Encoding code samples: <https://portswigger.net/web-security/cross-site-scripting/preventing>
24. Logical implementations of access control methods: <https://delinea.com/blog/access-control-models-methods>
25. Shamir's secret sharing: https://en.wikipedia.org/wiki/Shamir%27s_secret_sharing
26. Cryptographic shredding: <https://en.wikipedia.org/wiki/Crypto-shredding>
27. Definition of cryptographic salt: [https://en.wikipedia.org/wiki/Salt_\(cryptography\)](https://en.wikipedia.org/wiki/Salt_(cryptography))

28. Definition of cryptographic pepper: [https://en.wikipedia.org/wiki/Pepper_\(cryptography\)](https://en.wikipedia.org/wiki/Pepper_(cryptography))
29. Sec-Fetch headers: <https://w3c.github.io/webappsec-fetch-metadata>
30. CSP Strict-Dynamic: <https://speakerdeck.com/lweichselbaum/o-19-securing-web-apps-with-modern-platform-features?slide=23>
31. Details on how to complement CSP: <https://csp.withgoogle.com>
32. To test if your CSP is implemented correctly: <https://csp-evaluator.withgoogle.com>
33. To learn more about trusted-types: <https://github.com/w3c/trusted-types>
34. Do not use X-XSS-Protection and public key pinning: <https://www.invicti.com/blog/web-security/http-security-headers>
35. Same-origin policy: <https://portswigger.net/web-security/cors/same-origin-policy>
36. Use CORS to get around same-origin policy: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#:~:text=The%20same%2Dorigin%20policy%20is,documents%2C%20reducing%20possible%20attack%20vectors
37. HTTP security headers and secure cookies: *Alice and Bob Learn Application Security*, Chapter 2

Chapter 3: Improving

38. Database definition: <https://en.wikipedia.org/wiki/Database>
39. How to test and hardening your database server: *Alice and Bob Learn Application Security*, Chapter 6
40. File uploads: https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html
41. Privacy engineering: https://en.wikipedia.org/wiki/Privacy_engineering / <https://www.nist.gov/itl/applied-cybersecurity/privacy-engineering> / <https://arxiv.org/pdf/2007.08613.pdf>
42. Some tips on protecting files: <https://www.passportalmsp.com/blog/11-best-practices-for-document-management-security>
43. Nadeem Douba gave the workshop at OWASP Ottawa that taught me everything in that paragraph, winter 2014
44. Avoid bad functions: <https://www.freecodecamp.org/news/buffer-overflow-attacks>

45. Safe C libraries: <https://snyk.io/blog/buffer-overflow-attacks-in-c>
46. ASLR and DEP: <https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer-overflow-attacks>
47. Serialization and its uses: <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization>
48. *Alice and Bob Learn Application Security, Chapter 5*
49. OWASP Insecure Deserialization Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html
50. Privacy definition: <https://www.oxfordlearnersdictionaries.com/definition/english/privacy>
51. Quote: “Privacy is about self-determination and dignity. Not just data.” Abigail Dubiniecki, “Digital Self-Defence Post-Roe,” BSides Ottawa, November 2022
52. State of a system: <https://www.igi-global.com/dictionary/state-of-a-system/50412>
53. State of a system definition 2: [https://en.wikipedia.org/wiki/State_\(computer_science\)](https://en.wikipedia.org/wiki/State_(computer_science))
54. Review code for one system that delivers and handles all errors: https://owasp.org/www-community/Improper_Error_Handling
55. Log injection: https://owasp.org/www-community/attacks/Log_Injection
56. My boss, Bill Colls, taught me about locking records, way back in the day. I have approximated our conversation as it happened over 20 years ago. Bill, if you are reading this, thank you. You were my first professional mentor, and I have no doubt that your belief in me early in my career helped me get to where I am today. Thank you so much.
57. Cryptographic Storage - OWASP Cheat Sheet Series: https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html
58. Cryptographic Standards and Guidelines: <https://www.nist.gov/programs-projects/cryptographic-standards-and-guidelines>
59. Strongly typed languages: <https://www.techtarget.com/whatis/definition/strongly-typed>
60. What is DDD? Security article: <https://securityintelligence.com/posts/secure-coding-domain-driven-design>

61. Sec4dev 2022, “Typed Security Preventing Vulnerabilities By Design,” Michael Koppmann: <https://www.youtube.com/watch?v=TCc6gbTtK8o>
62. Reference, in general, to Eric Evans’ book *Domain Driven Design*

Chapter 4: Achieving

63. Reference to the book *Drift Into Failure*: <https://www.routledge.com/Drift-into-Failure-From-Hunting-Broken-Components-to-Understanding-Complex/Dekker/p/book/9781409422211>
64. Domain-driven design, from *We Hack Purple* podcast episode 68 with Gagandeep Singh Juneja: <https://wehackpurple.com/podcast/episode-68-with-gagandeep-singh>
65. Shostack’s 4 Question Frame for Threat Modeling: <https://github.com/adamshostack/4QuestionFrame>
66. “Securing Open Source Dependencies: It’s Not Just Your Code That You Need to Secure,” With Rana Khalil: <https://www.youtube.com/watch?v=oD0sDOvnwIc>
67. Paved road popularized by Netflix: <https://snyk.io/blog/three-considerations-building-security-program>
68. “Software developers are like water”: I have said this hundreds of times. Literally.
69. Paved road: <https://www.hella-secure.com/post/create-your-paved-roads>
70. Don’t forget licensing: <https://snyk.io/learn/code-security-audit>
71. Trusted system: https://wiki.owasp.org/images/a/ac/OWASP_SCP_Quick_Reference_Guide_v2.doc
72. OWASP Top Ten Security Misconfigurations: <https://owasp.org/www-project-top-ten>
73. OWASP XSS Filter Evasion Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html
74. Testing endpoint verification, OWASP Mobile Testing Guide: <https://mas.owasp.org/MASTG/tests/android/MASVS-NETWORK/MASTG-TEST-0021/#static-analysis>
75. Reusable code and why it’s awesome: <https://www.browserstack.com/guide/importance-of-code-reusability>
76. Reusing code in a proactive way: <https://datamyte.com/code-reuse>

77. System call definition: <https://www.geeksforgeeks.org/introduction-of-system-call>
78. Benefits and cons of commenting your code: <https://www.quora.com/What-are-the-benefits-and-drawbacks-of-code-documentation-versus-commenting-within-the-code-itself>
79. Consent is required: <https://auth0.com/docs/secure/data-privacy-and-compliance/gdpr/gdpr-conditions-for-consent#:~:text=According%20to%20Article%207%20of,withdraw%20consent%20at%20any%20time>
80. CRA class action lawsuit and privacy breach: <https://rhelaw.com/class-action/cra-privacy-breach-class-action-2>
81. OWASP Top Ten Software and Data Integrity Failures: https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures
82. Trust no one, not even your mom: *Alice and Bob Learn Application Security*
83. OWASP Top Ten Software Integrity Issues, how to avoid: https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures
84. Immutable definition, Oxford Languages website
85. More on immutable: <https://jamstack.org/glossary/immutable/#:~:text=While%20a%20%22mutable%22%20item%20can,is%20routed%20to%20them%20accordingly>
86. Brute force definition: <https://www.merriam-webster.com/dictionary/brute-force#:~:text=%3A%20relying%20on%20or%20achieved%20through,planned%2C%20or%20precisely%20directed%20methods>
87. Blocking brute-force attacks: https://owasp.org/www-community/controls/Blocking_Brute_Force_Attacks
88. NIST security control definition: https://csrc.nist.gov/glossary/term/security_control#:~:text=Definitions%3A,set%20of%20defined%20security%20requirements.
89. F5 security controls: <https://www.f5.com/labs/learning-center/what-are-security-controls#:~:text=Control%20Functions,-Preventative%20controls%20describe&text=activity%20from%20occurring.-,Examples%20include%20physical%20controls%20such%20as%20fences%2C%20locks%2C%20and%20alarm,%2C%20data%20classification%2C%20and%20auditing>
90. “Technical debt is security debt”: Dave Lewis of Cisco
91. Positive GRC story: Chris Carpenter
92. “Protecting Against Unexpected System Calls”: https://www.usenix.org/legacy/event/sec05/tech/full_papers/linn/linn.pdf

Chapter 5: Technology-Specific

93. SOAP losing popularity: <https://www.postman.com/state-of-api/api-technologies>
94. GraphQL not that popular: https://wundergraph.com/blog/graphql_rest_openapi_trend_analysis_2023
95. APIs are being attacked: <https://salt.security/blog/understanding-api-attacks-why-are-they-different-and-how-can-you-stop-them>
96. OWASP API Security Top Ten: <https://owasp.org/www-project-api-security>
97. API inventory: <https://owasp.org/API-Security/editions/2023/en/0xa9-improper-inventory-management>
98. WAF size: <https://curity.io/resources/learn/api-security-best-practices>
99. Auth N: <https://stackoverflow.blog/2021/10/06/best-practices-for-authentication-and-authorization-for-rest-apis>
100. More API sec best practices: <https://blog.hubspot.com/website/api-security#best-practices>
101. Google internal guidance, several points: <https://developers.google.com/maps/api-security-best-practices>
102. Changes to the OWASP API Security Top Ten: <https://nonamesecurity.com/blog/changes-to-the-owasp-api-security-top-ten-2019-to-2023>
103. Key management for APIs: <https://stripe.com/docs/keys>
104. All the mobile app security stuff, *NowSecure Mobile App Book*: <https://books.nowsecure.com/secure-mobile-development/en>
105. Use code signing certificates: <https://www.nowsecure.com/blog/2023/02/24/13-mobile-app-security-best-practices>
106. Memory is not actually erased on Android: <https://books.nowsecure.com/secure-mobile-development/en/coding-practices/understand-secure-deletion-of-data.html>
107. Avoid UUID: <https://books.nowsecure.com/secure-mobile-development/en/sensitive-data/limit-use-of-uuid.html>
108. Save your settings in your code: <https://books.nowsecure.com/secure-mobile-development/en/sensitive-data/protect-application-settings.html>

109. Do not include sensitive app data in phone backups: <https://books.nowsecure.com/secure-mobile-development/en/sensitive-data/avoid-storing-app-data-in-backups.html>
110. Disable the auto-correct feature for any sensitive information, not just for password fields: <https://books.nowsecure.com/secure-mobile-development/en/caching-logging/be-aware-of-the-keyboard-cache.html>
111. Avoid webview: <https://books.nowsecure.com/secure-mobile-development/en/webviews/prevent-framing-and-clickjacking.html>
112. Choose protection class carefully: <https://books.nowsecure.com/secure-mobile-development/en/ios/use-the-keychain-carefully.html>
113. Block caching of application snapshots: <https://books.nowsecure.com/secure-mobile-development/en/ios/avoid-cached-application-snapshots.html>
114. IOS memory safety features: <https://books.nowsecure.com/secure-mobile-development/en/ios/implement-protections-against-buffer-overflow-attacks.html>
115. Disable caching explicitly: <https://books.nowsecure.com/secure-mobile-development/en/ios/avoid-caching-http-requests-responses.html>
116. Secure Mobile Development Book: <https://books.nowsecure.com/secure-mobile-development/en/ios/implement-app-transport-security.html>
117. Avoid world-readable files on Android: <https://books.nowsecure.com/secure-mobile-development/en/android/implement-file-permissions-carefully.html>
118. Use intents carefully: <https://books.nowsecure.com/secure-mobile-development/en/android/implement-intents-carefully.html>
119. Content providers: <https://books.nowsecure.com/secure-mobile-development/en/android/implement-content-providers-carefully.html>
120. Protect app services from external callers: <https://books.nowsecure.com/secure-mobile-development/en/android/protect-application-services.html>
121. “Avoid Storing Cached Camera Images”: <https://books.nowsecure.com/secure-mobile-development/en/android/avoid-storing-cached-camera-images.html>
122. Avoid caching GUI objects: <https://books.nowsecure.com/secure-mobile-development/en/android/avoid-gui-objects-caching.html>
123. Sign your android APK: <https://books.nowsecure.com/secure-mobile-development/en/android/android-apk-signing.html>

124. Web socket definition: <https://en.wikipedia.org/wiki/WebSocket>
125. Web socket versus HTTP: <https://ably.com/topic/websockets-vs-http>
126. Web sockets: <https://developer.mozilla.org/en-US/docs/Web/API/WebSocket>
127. More web sockets: <https://datatracker.ietf.org/doc/html/rfc6455>
128. Cross-site WebSocket hijacking: <https://ably.com/topic/websocket-security>
129. Avoid tunneling: <https://devcenter.heroku.com/articles/websocket-security>
130. Max payload size and message format: <https://www.freecodecamp.org/news/how-to-secure-your-websocket-connections-d0be0996c556>
131. AWS Serverless Application Security Model: <https://aws.amazon.com/blogs/architecture/architecting-secure-serverless-applications>
132. Function-level security: <https://www.checkpoint.com/cyber-hub/cloud-security/what-is-serverless-security/5-serverless-security-tips-and-best-practices>
133. Function timeouts: <https://protectonce.com/the-10-best-practices-for-serverless-security>
134. ACLs and VPCs: <https://medium.com/@saurabhmarketer23/serverless-security-best-practices-ensuring-robustness-with-aws-lambda-311643eef268>
135. IOT from OWASP: https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project#tab=IoT_Top_10
136. Track IoT devices on network: <https://securityscorecard.com/blog/best-practices-for-securing-internet-of-things>
137. Endpoint protection: <https://www.fortinet.com/resources/cyberglossary/iot-best-practices>

Chapter 6: Popular Programming Languages

138. Devs love JavaScript: <https://radixweb.com/blog/top-javascript-usage-statistics>
139. What is JavaScript: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript
140. Typescript is translated into JavaScript: <https://www.toptal.com/typescript/typescript-vs-javascript-guide>

141. JavaScript invented: https://www.w3schools.com/js/js_history.asp
142. Avoid inline scripting: <https://sugarclub.sugarcrm.com/dev-club/b/dev-blog/posts/csp>
143. Avoiding DOM XSS: https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html
144. XSS prevention: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
145. Use strict mode: https://www.w3schools.com/js/js_strict.asp
146. DOM purify: <https://github.com/cure53/DOMPurify#dompurify>
147. Prototype injection: <https://www.synopsys.com/blogs/software-security/javascript-security-best-practices.html>
148. Subresource integrity checks: <https://raygun.com/blog/js-security-vulnerabilities-best-practices>
149. Avoid global variables: <https://aptori.dev/blog/javascript-security-best-practices-for-building-secure-applications>
150. What is CSS: <https://developer.mozilla.org/en-US/docs/Web/CSS>
151. Sensitive info in CSS: https://cheatsheetseries.owasp.org/cheatsheets/Securing_Cascading_Style_Sheets_Cheat_Sheet.html
152. Injecting into CSS: <https://www.geeksforgeeks.org/how-to-secure-cascading-style-sheets>
153. Avoid commenting in HTML: https://community.pega.com/sites/pdn.pegacom/files/help_v731/security/best-practices/sec-security-guidelines-custom-HTML-ref.htm
154. Dangerous HTML attributes and elements: <https://dev.to/ieeecsvitc/security-vulnerabilities-and-prevention-in-html5-p0m>
155. What is HTML: 5 <https://www.ericom.com/glossary/what-is-html5>
156. OWASP HTML 5 Security Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/HTML5_Security_Cheat_Sheet.html
157. Reporting bugs in PyPi: <https://www.python.org/dev/security>
158. Python best practices: <https://avatao.com/blog-common-issues-and-best-practices-in-python>
159. Using the virtual environment: <https://www.securecoding.com/blog/python-security-practices-you-should-maintain>
160. Avoid `eval()` and the like: <https://aptori.dev/blog/python-security-cheat-sheet-for-developers>
161. Use Python 3, not 2, etc.: https://res.cloudinary.com/snyk/image/upload/v1551385381/Python_Security_Best_Practices_Cheat_Sheet_.pdf

162. Bandit: <https://bandit.readthedocs.io/en/latest>
163. Use a linter and other Python best practices: <https://snyk.io/blog/python-security-best-practices-cheat-sheet>
164. *Alice and Bob Learn Application Security*, database section
165. SQL security best practices Microsoft: <https://learn.microsoft.com/en-us/sql/relational-databases/security/sql-server-security-best-practices?view=sql-server-ver16>
166. More SQL security advice: https://www.netwrix.com/sql_server_security_best_practices.html
167. Node facts: <https://nodejs.org>
168. Don't run node as root: <https://semaphoreci.com/blog/securing-nodejs>
169. Node.JS linters: <https://www.makeuseof.com/node-linters-top>
170. Node vulnerabilities: <https://nodejs.org/en/guides/security>
171. Node sec best practices, Snyk: <https://snyk.io/learn/nodejs-security-best-practice>
172. Node security best practices: <https://nodejs.org/en/learn/getting-started/security-best-practices>
173. OWASP Node Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html
174. Java usage: <https://thenewstack.io/java-usage-keeps-climbing-according-to-new-survey>
175. Oracle secure coding guidelines: <https://www.oracle.com/java/technologies/javase/seccodeguide.html>
176. *The CERT Oracle Secure Coding Standard for Java* by Fred Long
177. OWASP Deserialization Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html
178. Oracle Java SE security documentation: <https://docs.oracle.com/en/java/javase/14/security/index.html>
179. OWASP Java Security Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Java_Security_Cheat_Sheet.html
180. SEI CERT Oracle Coding Standard for Java: <https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java>
181. Typescript basics: <https://cloudblogs.microsoft.com/opensource/2018/03/05/typescript-overview>
182. Snyk TypeScript article: <https://snyk.io/blog/best-practices-react-typescript-security>

183. TypeScript Best Practices 2021: <https://medium.com/@warkiringoda/typescript-best-practices-2021-a58aee199661>
184. Aptori security TypeScript coding article :<https://aptori.dev/blog/secure-coding-in-typescript-best-practices-to-build-secure-applications>
185. TypeScript best practices: <https://github.com/andredesousa/typescript-best-practices>
186. Understanding the differences Between C#, C++, and C: <https://csharp-station.com/understanding-the-differences-between-c-c-and-c>
187. Use .NET features: <https://clickysoft.com/dot-net-security-best-practices>
188. Regular expression attributes and more: <https://www.syncfusion.com/blogs/post/10-practices-secure-asp-net-core-mvc-app>
189. All the C# specifics: <https://learn.microsoft.com/en-us/aspnet/web-forms/overview/older-versions-security/introduction/security-basics-and-asp-net-support-cs>
190. Logging in C#: <https://learn.microsoft.com/en-us/dotnet/core/extensions/logging?tabs=command-line>
191. Logging in .NET: <https://www.papertrail.com/solution/tips/7-best-practices-for-c-logging-with-examples>
192. C# security features: <https://clouddevs.com/c-sharp/cybersecurity>
193. NET auth namespace: <https://learn.microsoft.com/en-us/aspnet/core/security/authorization/introduction?view=aspnetcore-8.0>
194. HTTP requests: <https://learn.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-8.0&tabs=visual-studio%2Clinux-ubuntu>
195. Don't use secret management tool: <https://learn.microsoft.com/en-us/aspnet/core/security/app-secrets?view=aspnetcore-8.0&tabs=windows>
196. Anti CSRF defenses: <https://learn.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-8.0>
197. Razor encodes automatically: <https://learn.microsoft.com/en-us/aspnet/core/security/cross-site-scripting?view=aspnetcore-8.0>
198. Other auth types: <https://www.stackhawk.com/blog/guide-to-security-in-net>
199. Avoiding XXE: <https://www.syncfusion.com/blogs/post/10-practices-secure-asp-net-core-mvc-app>
200. C# corner: <https://www.c-sharpcorner.com/article/secure-coding-in-c-sharp-protecting-applications-and-user-data>

201. What is PHP: <https://www.php.net/manual/en/intro-what-is.php>
202. PHP security manual: <https://www.php.net/manual/en/security.php>
203. Escaping in PHP: <https://medium.com/@london.lingo.01/advanced-php-security-techniques-protecting-your-code-from-attacks-7d22ac5e2f74>
204. List of free PHP SAST tools: https://owasp.org/www-community/Source_Code_Analysis_Tools
205. Snuffleupagus module: <https://snuffleupagus.readthedocs.io>
206. PHP settings from Queens University: https://www.queensu.ca/its/sites/itswww/files/uploaded_files/Governance%20Instruments/queens-php-development-handbook.pdf
207. OWASP PHP hardening guide: https://cheatsheetseries.owasp.org/cheatsheets/PHP_Configuration_Cheat_Sheet.html
208. History of C: <https://www.linkedin.com/pulse/brief-history-c-evangelos-patsourakos>
209. History of C++: <https://www.oreilly.com/library/view/object-oriented-programming/9789332503663/xhtml/head-0045.xhtml>
210. *Secure Coding in C and C++* by Robert Seacord: https://insights.sei.cmu.edu/documents/1312/2005_009_001_52710.pdf
211. Avoid using void* and casting: <https://www.incredibuild.com/blog/top-10-secure-c-coding-practices>
212. SEI Cert C++: <https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-standard-2016-v01.pdf>
213. Compiler flags: <https://best.openssf.org/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++.html>
214. RAI: <https://blog.bytehackr.in/secure-coding-in-cpp>

Chapter 7: Popular Frameworks

215. What is Express? <https://expressjs.com>
216. Express.js security best practices, from the official source: <https://expressjs.com/en/advanced/best-practice-security.html>
217. Avoid vulnerable Regex: <https://github.com/davisjam/vuln-regex-detector>
218. Hydra: <https://github.com/vanhauser-thc/thc-hydra>
219. Express validator module: <https://express-validator.github.io/docs>

220. Checking JWTs: <https://www.freecodecamp.org/news/express-js-security-tips>
221. React best security practices: <https://relevant.software/blog/react-js-security-guide>
222. Redux.js package: <https://redux.js.org>
223. React.js best security practices: <https://snyk.io/blog/10-react-security-best-practices> (Note: this article was plagiarized or mostly copied in numerous other blogs, but this one was first, so I am referencing it.)
224. Good React libraries: <https://www.turing.com/kb/reactjs-security-best-practices#reactjs-security-best-practices>
225. Security-Realm: <https://www.freecodecamp.org/news/best-practices-for-security-of-your-react-js-application>
226. Official Angular Security page: <https://angular.dev/best-practices/security>
227. W3C school's definition of Trusted Types: <https://w3c.github.io/trusted-types/dist/spec>
228. Snyk Angular best practices: <https://snyk.io/blog/angular-security-best-practices>
229. Route guards: <https://www.scholarhat.com/tutorial/angular/tips-to-secure-your-angular-applications>
230. Vue.js advantages: <https://www.monocubed.com/blog/advantages-of-vue-js>
231. Official Vue.js security advice: <https://vuejs.org/guide/best-practices/security>
232. <https://vuejs.org/guide/scaling-up/ssr.html>
233. Vu.js: <https://medium.com/@wasantosfi/some-security-practices-to-vue-js-developments-e7512d07536d>
234. jQuery is old and outdated: <https://blog.jquery.com/2024>
235. "PSA jQuery is bad for security": <https://palant.info/2020/03/02/psa-jquery-is-bad-for-the-security-of-your-project>
236. Writing secure jQuery: <https://reintech.io/blog/securing-jquery-code-xss-vulnerabilities>
237. Secure coding guidelines from Microsoft for .NET: <https://learn.microsoft.com/en-us/dotnet/standard/security/secure-coding-guidelines>
238. .NET identity and authorization: <https://learn.microsoft.com/en-us/dotnet/standard/security/key-security-concepts>
239. Microsoft cryptography model: <https://learn.microsoft.com/en-us/dotnet/standard/security/cryptography-model>

240. OWASP .NET cheat sheet: https://cheatsheetseries.owasp.org/cheatsheets/DotNet_Security_Cheat_Sheet.html
241. Stackhawk .NET security guide: <https://www.stackhawk.com/blog/guide-to-security-in-net>
242. Overposting/mass assignment: <https://andrewlock.net/preventing-mass-assignment-or-over-posting-in-asp-net-core>
243. Official Ruby Security Guide: https://guides.rubyonrails.org/action_controller_overview.html#session
244. OWASP Ruby Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Ruby_on_Rails_Cheat_Sheet.html
245. Ruby security best practices: <https://www.codementor.io/ruby-on-rails/tutorial/ruby-on-rails-security-best-practices>
246. Spring Boot has replaced Struts: <https://www.orientsoftware.com/blog/spring-vs-spring-boot>
247. Spring basics: <https://spring.io>
248. Security, Spring Boot: <https://developer.okta.com/blog/2018/07/30/10-ways-to-secure-spring-boot>
249. Spring security integrating with cloud: <https://www.geeksforgeeks.org/best-practices-to-secure-microservices-with-spring-security>
250. Geeks for Geeks article on Spring security: <https://www.geeksforgeeks.org/spring-security-secure-your-web-application>
251. Cloud devs: <https://clouddevs.com/flask/best-practices-and-techniques>
252. Read-The-Docs Flask security: <https://flask-security-too.readthedocs.io/en/stable>
253. Flask security best practices: <https://python.plainenglish.io/the-best-security-practice-of-your-flask-application-api-691a21e37c27>
254. Flask security best practices: <https://medium.com/@sarah.schmidtdean/safeguarding-your-database-in-flask-e1a1302e3e33>
255. Best practices to protect your Flask applications: <https://escape.tech/blog/best-practices-protect-flask-applications/#owasp-top-10>

Chapter 8: Vulnerability Categories

256. Logic flaws: <https://portswigger.net/web-security/logic-flaws>
257. OWASP buffer overflow definition: https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

258. Wikipedia memory safety article: https://en.wikipedia.org/wiki/Memory_safety
259. Address space definition: https://en.wikipedia.org/wiki/Address_space
260. ASLR and other definitions: <https://www.fortinet.com/resources/cyberglossary/buffer-overflow>
261. Use GCC and other advice to avoid memory issues: <https://anchore.com/blog/breaking-down-nist-ssdf-compilers-and-interpretter-security>
262. Identity definition: <https://www.devx.com/terms/network-identity>
263. Transaction fraud definition: <https://www.datavisor.com/wiki/transaction-fraud>
264. Loss to ecommerce fraud: <https://www.statista.com/statistics/1273177/ecommerce-payment-fraud-losses-globally>
265. Replay attack definition: https://csrc.nist.gov/glossary/term/replay_attack
266. Prevention of replay attacks: https://en.wikipedia.org/wiki/Replay_attack
267. Crossing trust boundaries definition: <https://cwe.mitre.org/data/definitions/501.html>
268. Log4j CVE number: <https://nvd.nist.gov/vuln/detail/CVE-2021-44832>
269. SQLi CWE: <https://cwe.mitre.org/data/definitions/89.html>
270. Improper file handling definition, risk, and mitigation: <https://securityforeveryone.com/scan/parent/improper-file-process>
271. OOP definition from Reddit: https://www.reddit.com/r/computerscience/comments/lan2ssn/strictly_speaking_what_is_an_object_in_programming
272. Object-oriented programming and security: https://www.adacore.com/uploads/techPapers/Safety_Security_OOP.pdf
273. The difference between a class and an object: https://www.codecademy.com/forum_questions/558cd3fc76b8fe06280002ce
274. Polymorphism definition: <https://stackify.com/oop-concept-polymorphism/#:~:text=Polymorphism%20is%20one%20of%20the,types%20through%20the%20same%20interface.>
275. Race condition definition: <https://www.techtarget.com/searchstorage/definition/race-condition>
276. Race condition causes: <https://www.indusface.com/learning/what-is-a-race-condition-vulnerability>

- 277. Race conditions allow you to exceed a business logic limit: <https://portswigger.net/web-security/race-conditions>
- 278. More on preventing race conditions: <https://www.imperva.com/learn/application-security/race-condition>
- 279. Unreleased resources: <https://cwe.mitre.org/data/definitions/404.html>

Chapter 10: Design

- 280. Definition of threat modelling from OWASP: https://owasp.org/www-community/Threat_Modeling
- 281. Adam Shostack's four-question frame for threat modeling: <https://github.com/adamshostack/4QuestionFrame>
- 282. Data flow diagram definition: <https://www.lucidchart.com/pages/data-flow-diagram>
- 283. User stories coming into existence: <https://www.knowledgehut.com/blog/project-management/3-c-of-user-stories>
- 284. OWASP security user stories: <https://github.com/OWASP/user-security-stories/blob/master/user-security-stories.md>

Chapter 11: Coding

- 285. Definition of code review: <https://www.synopsys.com/glossary/what-is-code-review.html>
- 286. When to upgrade dependencies: <https://softwareengineering.stackexchange.com/questions/340705/when-should-dependencies-be-updated>

Chapter 12: Testing

- 287. Definition of testing coverage: <https://testsigma.medium.com/test-coverage-in-software-testing-a-comprehensive-guide-8b3d0386a0a7>
- 288. Depth versus coverage: <https://www.packetlabs.net/posts/coverage-based-penetration-testing>

Chapter 13: Release/Deployment

- 289. OWASP CI/CD Top Ten: <https://owasp.org/www-project-top-10-ci-cd-security-risks>
- 290. OWASP Top Ten A08:2021 – Software and Data Integrity Failures: https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures

Chapter 14: Maintenance

- 291. Monitoring software definition: <https://www.techopedia.com/definition/4313/monitoring-software>
- 292. Why we monitor software: <https://www.techtarget.com/searchitoperations/definition/IT-monitoring>
- 293. Security alert definition: <https://cyberpedia.reasonlabs.com/EN/security%20alerts.html>
- 294. Difference between monitoring and observability: <https://aws.amazon.com/compare/the-difference-between-monitoring-and-observability>
- 295. WAF definition: <https://www.cloudflare.com/en-ca/learning/ddos/glossary/web-application-firewall-waf>
- 296. Content delivery network definition: <https://www.cloudflare.com/en-ca/learning/cdn/what-is-a-cdn>
- 297. CDN security: <https://www.sentinelone.com/cybersecurity-101/what-is-a-cdn-content-delivery-network>
- 298. RASP definition: https://en.wikipedia.org/wiki/Runtime_application_self-protection
- 299. API gateway definition: <https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html>
- 300. IT security incident definition https://csrc.nist.gov/glossary/term/security_incident
- 301. Business continuity definition: https://en.wikipedia.org/wiki/Business_continuity_planning
- 302. Disaster recovery definition: <https://cloud.google.com/learn/what-is-disaster-recovery>

Answer Keys

Chapter 1: Introductory Security Fundamentals

1. We retest our code after we have fixed a bug to ensure we have not unintentionally created a new problem.
2. We keep our frameworks up to date so that we are on a supported version, so that it is easier to upgrade later, so that we get receive features (including security features), and so that we receive security patches (fixes for security problems).
3. I like the new OWASP DSOMM framework. I take the ideas I like the most and apply them, but leave the rest. The options it gives are quite versatile; they seem to fit with most modern SDLCs.
4. Part of my threat model is very similar to other women; we have risks that we have to look out for when it comes to spending time with men whom we do not know and trust. On top of this, as a person with many admirers on the internet, I have extra threats, and have had more than one stalker in my life. As a result, I need to protect the location of my home (from uninvited visitors), my professional reputation (from slander and impersonation), and even my domain (I've had 'admirers' report `wehackpurple.com`, my old company's domain, as malicious, over and over again, resulting in a

loss of business). As a person who makes most of her living off of the intellectual property she creates, I also need to guard against plagiarism, which is an incredibly difficult battle. The list of threats is (unfortunately) quite long if you have a public persona.

5. Whenever I click “no thank you” to cookies on a company’s website I assume that they will respect that choice. That’s me, assuming I can trust their marketing team not to track me across the internet without my consent.

Another example of implied trust is when I apply for a conference. I submit my research to the conference, and assume they will read it and evaluate it fairly for potential entry to their conference. But what if one of the people who sees my research decides to change it a bit, and then submit it to another conference, with their own name on it? How would I know? How would I prove such a thing? I do not think anyone has done this, but it’s a threat few of us think of. Most of us blindly assume trust many times a day.

6. I learned several new ways to work with GRC (governance, risk, and compliance) teams in the writing of this book, and I plan to use those new ideas (sharing the information) going forward. Especially when I experience friction when working with such teams.
7. When I set up the password manager for my entire company (We Hack Purple), I created different vaults for different teams. I had one set of passwords that the marketing team needed, such as social media accounts. I had another set for the administration of the company, which my admin assistant and I could access, but no one else. This made it very easy to rotate credentials (reset every password) whenever someone left, as I knew exactly what they did, and did not, have access to. If least privilege was not applied in this way at the company, it would have been possible for someone to access different parts of the company that they should not. For instance, the administrative assistant could have tweeted from the company account “I don’t even like purple!”, and that clearly would not be good. I would also have had to reset 100% of passwords every time someone left the company, which would not have been a productive or fun use of my time.
8. A security library that contains security control functions (input validation, output encoding, encryption, etc.), would be an excellent secure default. “Use the security library to implement every security control possible, do not write your own.” Then it all becomes easy!.
9. When a company puts APIs onto the internet, without an API gateway, and then thinks that means no one can see them, that is an attempt at security by obscurity. Yes, you have not announced they are there to be found, but it is short work for someone to find them.

Another example of security by obscurity is not sharing a lot of personal details about yourself online, such as where you live, who your family members are, what your work pass looks like, your birthday, etc. The more information you give away, the easier it is to for someone to steal your identity, impersonate you, or answer your security questions on any given website you shared correct answers on.

Note: don't give accurate answers for security questions. Instead, give long, random, unique values instead, just like you would for the password field. Then save them, along with the questions, into your password manager. This is much more secure than (1) sharing personal data with random sites you make accounts on and (2) choosing something someone could guess easily after looking at your Facebook page for 30 seconds.

10. If you have a shopping cart as part of your application, if there is an error as someone is checking out, you should definitely roll back and fail closed. What if you charged them but didn't give them what they purchased? Or the reverse? Or what if they figure out a way to abuse this error in your checkout so that they can pay once, but receive multiple products? There are many ways any sort of transaction can go wrong if we attempt to continue after a system error; don't be fooled into thinking that it's easier, it's not.

Chapter 2: Beginning

1. False. All input needs to be validated. What if that API is being called from somewhere else? What if your app has been compromised? What if your app has poor input validation and it missed something?
2. Once. Unless you've done a really bad job.
3. AuthN and then AuthZ. It's important because you need to know who someone is before you can let them into your systems. We don't want them to be even one step inside our system until we know they are a recognized user.
4. API keys, connection strings, passwords, certificates, hashes, API secrets, license keys
5. True
6. False
7. Attackers can use that information to form better attacks. It can be confusing to users, and it makes us look unprofessional and reflects poorly on our application.

8. Attackers can use errors to force our application into an unknown state and then attack. It can be confusing to users, it makes us look unprofessional and reflects poorly on our application, and it may result in something completely unexpected, good or bad.

Chapter 3: Improving

1. Yes. Of course there is! Often it is not cost-effective to re-write an entire app. Or it's used on a closed network, so the likelihood of there being someone malicious on the network is so incredibly low that an outsider being on the network is more of a risk than an overflow vulnerability would be (because you already have remote access to the servers at that point). You don't always have to use memory-safe languages, but unless there's a reason not to, you should.
2. False! You can make mistakes (bugs) in both strongly and weakly typed programming languages!
3. Both!
4. True! Please don't block pasting into the password field of your apps.
5. True. Please advocate for the privacy of your users, fellow citizens, and everyone else whenever possible.
6. True! We must protect the server, the database software, the network, and the data inside the database itself.
7. False. Come on!
8. This is a trick question! It is true that it's intellectual property, and it's true that you need to protect its integrity and availability, but you don't *always* need to protect its confidentiality. If you work on an open-source project, you have decided to make your code public. That means you still own the copyright and can license it however you (and your other contributors/project members) want. But you've decided, actively, not to protect the confidentiality of the code itself if you make it open-source. It still has value, but you're open to sharing it with the public. That said, you still want to ensure that (1) no one deletes it (that would be awful) and (2) no one negatively affects its integrity (for instance, adding code with intentional security issues so it could be exploited later). All of that said, the answer is still "true," whether you have made your source public or not. Take care of your code!
9. This is true, but there's more to it. You can use serialized objects that you have created (and by that, I mean serialized by systems you/your org control and know for a fact can be trusted and that the serialized object

has zero possibility of being manipulated or swapped out by a malicious actor). The key takeaway here is not to deserialize any object from an untrusted source, and an untrusted source can be from your own systems if they leave any opportunity open for your serialized objects to be accessed, changed, or manipulated by a third party. When in doubt, don't deserialize!

10. False. Whenever I am told that people "trust their employees" as a reason for why it's okay not to perform basic security hygiene, it makes me think they just don't understand how human beings work. Would you take out the millions of dollars your company has in the bank, walk into the break room in your office, plop it down on the nearest table, and then go home for the day? No? Why not? But. . . I thought you trusted your employees! See how that isn't a real answer? Production data should only be kept in production, and masked versions of it (or, conversely, made-up test data) should be stored and used for lower environments and testing in lower-level, nonproduction environments.

Chapter 4: Achieving

1. An example of technical debt would be using out of date libraries or frameworks. Updating your frameworks and libraries are ways to repay that technical debt.
2. Protect your work accounts by using long, unique passwords, a password manager, and enabling MFA. We protect our work accounts so that we do not lose our jobs, endanger our organization, and because as superusers we are the targets of malicious actors.
3. Input validation, output encoding, scanning uploaded files for viruses, authentication, authorization, policies, video cameras, guard dogs, etc.
4. Brute force attacks are usually used to break into systems (forcing their way past authentication and authorization), which means malicious actors are getting into your systems they should not be in, and that is terrible in anyone's opinion. It is difficult to protect against such attacks for several reasons, users tend to reuse passwords and select weak passwords, stolen credentials are sold cheaply (and illegally) on the internet, and there are automated attack scripts and bots all over the world, ready to attack your apps at a moment's notice.
5. If a software's integrity has been compromised every action it takes may be incorrect, including giving you the incorrect total when you go to the checkout, or the wrong amount or type of medication if gives you. The results can be deadly, literally.

6. Users are the judge of our work. If they do not like it, they will not use it (unless legally required to do so, or forced by their employer). As such, they will go to your competitors if they do not like the services, features, and terms you provide them. This is one reason. Another reason is that we must legally obey a user's wish in some cases, and have their consent as a result of legislation, such as GDPR in Europe.
7. Documentation means we know what an app does, how it's built, and that will help us respond when there is a security incident or we need to update it.
8. Comments tell the next person, who may be us, what each part of the code does, which helps us fix bugs and respond to security issues faster.
9. We need to protect system calls from injection (among other attacks), to avoid malicious actors being able to control our systems. We cannot allow control to fall into the wrong hands.
10. Benefit: Save time and money. If it's already been security tested this means you know it's safe to use, and will not have security bugs in it.
Con: If you have not tested your code, there is a security bug in it, and then you reuse it, you now have two or more security bugs.
11. If we do not harden our systems (select safe configuration choices) they could be successfully attacked, easily. Systems that are purchased and popular (such as Jira or Confluence) could have known security weaknesses if you make poor configuration choices, meaning a non-advanced attacker could potentially successfully exploit your systems. An example of hardening a system would be changing the administrative username to something else, changing the default password to something long, complex, and unique, enabling MFA, turning on logging and monitoring.
12. An allowlist is a list of characters (letters, numbers, special characters) that are allowed to be accepted as input for a variable, system or parameter. A blocklist is the opposite, it is a list of characters you believe to be bad, that you want blocked. It is extraordinarily difficult to make a perfect blocklist, but making a perfect allow list is usually rather easy.
13. A trusted system is one that we (the IT department / owner of the system) controls, and cannot be changed/adjusted/modified by an outsider. This generally means the server, rather than front-end JavaScript, which can be manipulated in the web browser or from a web proxy tool (sometimes part of a DAST). Because the requests and responses to and from the browser can be manipulated, they cannot be trusted, and thus they could be easily attacked. We must perform all important functions on a trusted system to ensure their integrity is not violated.

14. If your code is unreadable then it's difficult to review (for any type of issue, including security issues), it's difficult to fix bugs in, and add new features to. Unreadable code will make everything take longer, and thus cost more, whenever you work on it.
15. A secure default is when the original value for a setting or choice is set to a secure choice, rather than a more open and/or less secure choice.

A paved road is one or more security settings or choices that are set out for users or software developers that ensure the easiest way to do something is also the most secure way. It's the path the security team want you to follow when you are using a system or performing an action, and they design it such that you must use extra effort in order to perform that action in a less-secure manner.
16. Secure Default: having MFA enabled by default on new accounts, instead of making the users go into a sub menu to turn it on, after they have created the account.

Paved road: creating a template in your programming language for all new applications that has input validation, output encoding, and other security features built into it. It's easier to use the security features provided, than writing your own from scratch.
17. Yes, if you use a CI/CD to build your software it is part of your supply chain. Everything you use to build software is part of your supply chain.
18. In an ideal world we would never use dependencies with vulnerabilities in them; we would all use perfectly secure libraries. But we do not live in a perfect world, and that would be both extraordinarily expensive, time consuming, and limiting. As a result, I recommend that people upgrade off of or remove any dependencies that have *reachable* or *exploitable* vulnerabilities, and if do not know if the dependency's vulnerability is reachable or exploitable then I assume it is vulnerable and apply the same rules that I do to other vulnerabilities in prioritizing its removal or upgrade. Reachability can help you reduce the number of issues you are looking at by over 90%, it's a huge time saver.
19. Ideally, we would threat model during the design phase of a project, but I will happily take a late threat model over no threat model. That said, in the design phase you have enough information to do a great threat modeling session, but you haven't built anything yet, which means it's free (or almost free) to make changes to the design at that point. It also the easiest time to make huge changes to the design, so that you are not tacking security on later, which often proves less than perfect.

Chapter 5: Technology-Specific

1. Bots and automated types of attacks of any other kind, potentially called scripts.
2. Yes: you can use them for serverless apps as well!
3. *Of course we do!* The normal, secure coding practices apply to *any* technology that includes code.
4. False! That information could be used by a malicious actor to form a syntactically correct attack.
5. Yes. Perhaps even more than your web apps if the web apps don't contain any business logic. Your APIs are all brains, performing business logic decisions all day long; it is very important that you test them thoroughly.
6. Because malicious actors are looking for them! Older apps, API or otherwise, are often not properly cared for and security tested and thus can be full of security vulnerabilities. Unless you have a business requirement to keep the old version on and maintained, turn off unused/no-longer-supported APIs (and any other systems that fit that description!).
7. Mobile apps are often decompiled by attackers, significantly more often than other types of applications. For APIs and (most) web apps, the code lives on the server and is therefore not available for the attacker to grab a copy and decompile. Not true with mobile apps, where they have the device that it runs on: the phone!
8. Phones have NAND flash memory that is constantly trying to cache information, plus the key chain, autocorrect, automated app backups, and more, all of which might save sensitive data in places they shouldn't. We need to take several extra precautions when it comes to securing sensitive data on a phone.
9. This is an application of the least privilege and assume breach principles. We want to reduce the amount of permissions we ask for in case something bad happens. If our app is successfully attacked but we have a very small set of permissions, the damage an attacker can do is greatly limited, making the attack less terrible.

10. On Android, sometimes the pointer is removed but the actual data is not replaced with something else in memory. As a result, a malicious actor could take the phone, go through the unallocated memory, and find your sensitive data. Oh no!
11. If our app contains sensitive data, we should be careful that data is not part of the phone app data backup, which would not protect the sensitive data the way we need it to by default.
12. No, it's not! Neither is Android! This means we must take many extra precautions to avoid overflow types of vulnerabilities. Be very careful, folks; this is a serious type of bug!
13. It's two-way/bidirectional. It can even send messages both ways at the same time.
14. No: they use WSS (Secure Web Socket Protocol), WebSocket's own version of encryption in transit.
15. This is an application of least privilege; if we use tunneling and we end up vulnerable to XSS (which is extremely common and happens often), our tunnels will be accessible to the attacker! Not using tunneling reduces our attack surface greatly.
16. Transactions should only take a certain amount of time. If one specific transaction takes way longer than usual, that is sometimes indicative of an attack. Creating a timeout can stop an attack midway through, assist with rate limiting, and save money.
17. According to the OWASP API Security Top Ten (2023), 4 of the top 10 issues for APIs are access control. Attackers are attacking various levels of broken access control and succeeding more now than ever before. We must check *every single function call*.
18. Unfortunately, malicious actors are creating malware to attack IoT devices, and endpoint security products can protect us from some of those attacks.
19. Network segmentation is an implementation of assumed breach, and with IoT, it's no different: we want to design and plan as though one day we will be breached. IoT devices are targets on our networks, and if we can segment them separately, if one gets attacked successfully at least only a small part of our network is exposed.

Chapter 8: Vulnerability Categories

2. A business logic flaw is when the program does something unexpected and not what it is supposed to do. Because it's a flaw, not a bug, it's an error in the decision the app is making, not the way it is coded. Code scanners can find code problems, but they do not understand the decisions the app is trying to make. They can't read minds, to understand that the app is not following the plans you intended for it.
3. Someone uses a coupon to get 50% off of a pair of pants. The coupon code was "october2025", and the current month was October. The next month they buy more pants and use the code "november2025" and it works again. The person has abused a pattern of discount codes, and harms the business, who was only expecting the code to be used by special customers, not just anyone.

Someone is playing on online game and plans to purchase a virtual weapon within the game. They open the game on two different computers, sign into the game on both machines, then perform the same transaction on both machines at the same time. Somehow, after the transaction they receive two of said virtual weapon. The people who are hurt here are the gaming company as well as other players, as the perpetrator here has an unfair advantage in the game, and the gaming company misses out on additional in-app purchase opportunities.

4. I once pentested SharePoint, and found XSS all over the place. I was able to store the XSS in the names of pages, and then whenever someone visited the page it attacked them. They had not followed the hardening guide, which fixed the issue immediately.
5. Authentication is "who are you?" and authorization is "should you even be here?". I was going to write that you could not have one without the other, but I am thinking of security only. From a security perspective you need both. But let's say you're marketer who wants to spy on people and using retargeting ads to follow people all over the internet. They could authenticate visitors (figure out who they are), and then track them, but not grant them access to anything. So, I suppose you can use one without the other, if anyone from the public is allowed to access said resource.

6. I think secrets are difficult to manage for several reasons. 1) Secret management tools are extremely costly, and that means not everyone can afford to use them. 2) Most of us were taught to store secrets in our code, and so we are all trying to unlearn that lesson. 3) Mistakes happen.
7. Falling into an unknown state is the one I personally find most disconcerting, because then I do not know what will happen. With a race condition or injection, I know it will be terrible, but I can also guess which thing(s) will happen. With falling into an unknown state all bets are off.
8. I'm going to use a memory safe language and avoid complexity whenever I can!

Chapter 9: Requirements

1. I would like to add requirements for a threat model, a SAST scan, a secret scan, and a dynamic scan to my project, as a minimum. I would also like to meet the security person during the kick off meeting so I can ask questions.
2. Web App: all traffic encrypted in transit and at rest.
API: accessible only from the API gateway, and the following features are enabled, rate limiting, authentication, authorization, and input validation.
Database: No system accounts have database owner privileges, and the database can only be accessed from the app via parameterized queries.
3. All requirements for any project need to be documented and agreed to, if you want to be sure they are part of the final product. If it is not a requirement, you may not get it.
Security requirements are not obvious to most IT workers, as they are not taught in most schools or training programs, unless the program is focused on security as a topic (such as a secure coding course). General project management and software development courses do not cover security at all, or only in a minor way.
4. Project kick off meetings introduce all key stakeholders, identify many expectations, responsibilities are determined, and it officially announces that the project has started. Every single one of those things is extremely important for a successful project.

5. When I was a software developer, some of the questions I asked included:
 - Are we going to have a pentest at the end? And is that coming out of my project's budget? I would prefer it comes out of the security budget.
 - Will you arrange the pentest for me? Or do I need to find someone myself and do procurement? Do you have an existing contract vehicle?
 - Are there any automated tools you're going to run on my code/app? If so, may I have access to them so I can scan it first and fix it myself?
 - May I have any forms you're going to want me to fill out now? So I know what you will need me from? Such as an architecture review document?
6. If the security activities for a project are not part of the schedule, when will you perform that work? Evenings and weekends! We need to include the time for each security activity, and additional time to fix any issues that arise (such as test results), in the schedule so that we can have better work-life balance, finish our projects on time and on budget, and to avoid employee burnout and staff turnover.

Chapter 11: Coding

1. If you point a dynamic automated scanner with a fuzzer (generally referred to as DAST) at an application you are attacking that web app. When you hit the 'scan', 'attack', or 'active scan' button, you are attacking whatever you have aimed it at. In most countries, this is illegal, and I do not condone anyone committing cybercrimes. You could easily break someone's API or web application, ruin their data, or perform a denial of service attack. Please do not test an app with a DAST unless you have written permission to do so.
2. Yes. You can scan any code that you can get your hands on with a SAST and you will never hurt a thing. They are non-destructive, all they do is read the code and analyze it for vulnerabilities. If you want to be an extra-wonderful person, send a report or file issues for the open-source project with the bugs you found.
3. If we do not keep our dependencies up to date we will accrue technical debt, we will not have the latest cool features, and we will not have the latest security updates. It also makes it more difficult to upgrade off of a vulnerable version quickly, if a serious vulnerability is being exploited actively in the wild.

4. I currently have Docker, Semgrep, and 42Crunch installed in VS Code.
5. A secure guardrail is a technical control that nudges someone away from doing something insecure/dangerous, and towards doing something safer. Guardrails on roads help keep cars *on the road* (as opposed to off the road, which is not as safe). Setting a secret scanner to scan all new code for secrets in a pre-commit hook for your version control repo, which rejects anything that looks like a secret and issues a warning, would be a great secure guardrail. Another example would be if you took a list of all the ‘functions you should avoid’ from chapter 6 for your language of choice, and then made a SAST rule to find each one and alert you. Running those SAST rules in your IDE and then also when you check your code in would be another great secure guardrail.
6. We review code to improve security, quality, readability, style, etc. The benefits are everything previously mentioned, plus we learn to write better code. Win!

Chapter 12: Testing

1. As a tester, if we break something that is not in scope, we could get sued. If we hire someone to do a test for us, and we do not set clear scope, they may attack things we do not want attacked, and crash the system, ruin our data, see data they shouldn’t, etc.
2. We need a signed document so that it can be used in court if we get sued. Penetration testers and security researchers used to get sued all the time. Having a signed document with a clear scope makes it very easy to know if someone did, or did not, act inappropriately.
3. Penetration testing is done by an expert, whereas anyone with the tiniest amount of skill can run a scan. Penetration tests include many tools, and manual testing methods, with the goal of finding every possible security issue, and then testing to see if they are exploitable. An automated test is not always able to truly validate its own results, none of the tests are manual, and they tend to find a lot fewer things that a talented pentester.
4. True. They can ruin your data, create a denial of service situation, or even take down your entire web server. Any DAST that sends requests to your system (instead of just passively reading responses) can cause damage.

Note: it is possible to run a DAST only in passive mode, but the results are extremely limited.

5. A combination of both automated and manual testing will always provide the best results. Getting both sides of the story is always better.
6. False. The purpose of fuzzing is to test your input validation.
7. A good penetration tester is an incredibly powerful person, put into a very trusting position, and thus, they must have good ethics. You give them the keys (credentials) to many systems, allow them to set up dangerous tools inside your network, and then allow them to test the limits of your systems. On top of this, often when we hire a penetration tester we do so because we ourselves are not an expert in this area, so we have no idea what they are doing. We also tend to pay pentesters rather well, and allow them to do their work remotely. This is a huge amount of trust to place in a stranger. If a penetration tester does not have good ethics, they can harm your organization in many, many ways. They could break your IT applications, access sensitive data or systems, or commit fraud by pretending to do a penetration test but only running a simple automated scan. It is absolutely imperative that penetration testers have ethics that are beyond reproach.
8. Similar to a penetration tester, IT workers are very powerful, especially software developers, system administrators, and the folks from help desk. We have access to all sorts of amazing systems, data, and privileges. If an IT worker becomes an insider threat (an employee or contractor who causes harm, usually intentionally, to an organization they work for), they are often the most dangerous because they have the most access. Please practice good ethics. If you are tempted to do something you know is wrong, perhaps it's time to go work somewhere else?

Chapter 13: Release/Deployment

1. The CI/CD is part of your supply chain AND an extremely powerful tool. We must secure it so it is not misused (either intentionally or unintentionally). Unauthorized users should not be able to release code, delete reports, deploy containers, remove or disable important tests, etc.
2. Signing your code guarantees its integrity (it has not been changed since you created that copy) and authenticity (that it came from your team/org as opposed to from an attacker).

3. Since 2020 the public has been aware of attacks where malicious actors have replaced, injected code into, or otherwise changed a legitimate 3rd party library such that it became dangerous. Then the same threat actors attacked applications that used the affected library. There are also cases where people accidentally downloaded the wrong version of libraries, either due to malice or error, which also had poor consequences. These new attack vectors have made it very clear that we must validate the integrity of each dependency we use in our applications.
4. See answer 1. Someone malicious with access to control your CI/CD could release malicious code, disable security tests, run up your cloud bills, and waste a lot of resources, to name a few things that could go wrong.
5. I would like to see SAST, SCA, IaC (infrastructure as code) and secret scanning in the CI/CD.
6. I don't generally put DAST into the CI/CD unless I'm trying to prove a point or show off. I find it to be a ton of work for very little gain, in my experience. I also wouldn't do anything else that is overly resource intensive (DOS, stress, or performance testing) or that could cause damage (Metasploit anyone?) in my CI. Static tools (code analysis) are better for the CI than dynamic (interacting with the running app) as a general rule.
7. Secret scanning is my favorite because it's so easy to do and if you find a live secret you probably just avoided a security incident. When I find a really big secret it makes me feel brilliant; even if the tool did all the work, I just saved the day.

Chapter 14: Maintenance

1. If your software is not part of the DR plan that means when a disaster happens your part of the organization will not be ready, and you will let the rest of the business down. I've worked at places that had disasters, and some of us could keep working, while others had no plan at all. Disasters are more common than you may realize, and being ready is one way to shine bright when it comes to promotion time.
2. DR covers mostly the IT department and systems used to support the business, with the goal to have the business back up and running at a bare minimum level, as soon as possible. If your business sells widgets, that means it needs to be able to sell widgets again. If it provides services, it should be able to provide at least some of the services again, as soon as possible.

BCP covers everything in the business, and getting back to normal (full) operations, as soon as possible (which will take longer than DR). BCP covers people, where you work, processes, and IT.

3. Need to know means only sharing information with people who have a requirement to know the information, as opposed to curious people who would like to know. If someone is part of an incident investigation, they likely need to know all about it. If you overhear something in an elevator that sounds like juicy gossip, you do not need to know.

Need to know is important to apply when responding to security incidents because 1) the information you know could be potentially damaging to your business, customers, citizens, or employees if shared, 2) the incident may involve an insider threat and you might tip them off, or 3) it could cause a huge panic and scare people. There are many more reasons to not gossip about sensitive topics, these are just a few.

4. We continue to test because testing tools regularly receive updates with new types of tests, new vulnerabilities are found in libraries and frameworks all the time, and because we may have previously missed something.
5. We have multiple layers of defense in case one fails, or someone is able to circumvent (go around) one of our layers.
6. If you write code as part of your job, you need that code to be secure, and that includes you read reader. Data scientists handle our most precious digital resources: our data! We need them to protect it as best they possibly can.

42Crunch IDE plugin, 132

A

AAA (authentication, authorization, accounting/auditing), 31
 access control, 32–33
 ACLs (access control lists), 34
 CI/CD, 300
 DAC (discretionary access control), 34
 file management, 60
 MAC (mandatory access control), 33
 models, 33–34
 RBAC (role-based access control), 33, 34
 secret management, 35–37
 session management, 34–35
 access issues, 225
 account takeover, 289
 ACLs (access control lists), 34
 group policies, 34
 AD (Active Directory), identity and, 32
 Adam Shostack's Four Question Frame for Threat Modeling, 260
 address space, 217
 administrative security controls, 110
 AI (artificial intelligence), safe use, 325–327
 alerting, 72–73
 logs, 307
 metrics, 307
 traces, 307
 allowlists/approved lists, 97–98
 Angular, 186–190
 API gateways, blocking and, 310–311

APIs (application programming interfaces), 127
 best practices, 127–133
 DPAPI (Windows Data Protection API), 195
 gateways, 4
 scoping and, 282
 testing and, 284
 AppSec, 289–290
 ARC (Automatic Reference Counting), 136
 architecture whiteboarding, design and, 263
 ASLR (Address Space Layout Randomization), 136, 219
 assume breach, 9
 ASVS (Application Security Verification Standard), 14
 ATS (App Transport Security), 136
 attack surfaces, reducing, 11
 Attack Trees, 260
 attacks
 brute-force, 109–110
 credential stuffing, 38
 DOS (denial-of-service), 109
 man-in-the-middle attacks, 100
 manipulator-in-the-middle attack, 100
 replay attacks, 228–229
 timing attacks, 237
 auditable code, 93–96
 authentication, 31–32
 2FA (two-factor), 57
 MFA (multifactor authentication), 57
 authentication issues, 223–224
 AuthN (authentication), 31
 authorization, 31
 access control, 32–33
 authorization issues, 225

AuthZ (authorization), 31
 automated testing, 286–287

B

BA (business analyst), 98
 backward compatibility, 16
 BCP (business continuity plan), 9, 315–317
 best practices
 AI use, 325–327
 APIs, 127–133
 good habits, 319–322
 IoT (Internet of Things), 140–141
 mobile applications, 134–137
 serverless computing, 138–139
 WebSocket, 137–138
 blind SQL injection, 222
 blocking, 308
 API gateways, 310–311
 CDNS (Content Delivery Networks), 309–310
 data scientists and, 311–312
 RASP (runtime application self-protection), 310
 virtual patching, 310
 vulnerability shielding, 310
 WAFs (web application firewalls), 309
 blocklists, 97, 98
 BOLA (broken object level access control), 234–235
 bounty hunters, 289
 breaking the build, 297–298
 brute-force attacks, 109–110
 buffer
 overflows, 216–217
 overreads, 217
 bugs, 82
 bounty programs, 289–290
 causes, 215
 developers fixing, 292
 false positives, 292
 preventing, 215–216
 retesting fixes, 18
 risk and, 215
 Burp Suite Pro, 277
 buy, borrow, build, 224

C

C#, 166–170
 catching errors, 69
 C/C++, 175–178
 CCPA (California Consumer Privacy Act), 14, 41
 CDNS (Content Delivery Networks), 309–310
 chapter exercise answers, 363–367
 ChatGPT, 325–327
 Checkmarx SAST, 271

CIA (confidentiality, integrity, availability)
 traid, 4–6
 DFD (data flow diagram) and, 263
 CI/CD (continuous integration/continuous delivery), 295
 access control, 300
 breaking the build, 297–298
 DAST (dynamic automated scanners), 296–297, 298
 dependencies, 300
 fuzzing, 297
 IaC scanners, 299
 linting, 299
 package integrity, 300
 performance testing, 297
 pipeline security, 299–302
 SaaS (Software as a Service) products, 301
 SAST tool, first generation, 296, 298
 SCA (software composition analysis), 299
 secret scanning, 298
 stress testing, 297
 CIS (Center for Internet Security), 15
 clear box testing, 270
 cloud providers, temporary credentials, 58
 code
 auditable, 93–96
 bugs. *See* bugs
 readable, 93–96
 reusable, 100–101
 code review, 270
 black box testing, 277
 Burp Suite Pro, 277
 clear box testing, 270
 DAST (dynamic application security testing tools), 276–277
 dynamic scanners, 277
 SAST (static application security testing) and, 271
 static analysis, 270
 first generation tools, 271
 free tools, 271
 second generation tools, 271
 secure guardrails, 272
 white box testing, 270
 Zap (Zed Attack Proxy), 277
 code signing, 107–109
 coding
 dependency verification, 274–275
 IDE plugins
 IaC scanner, 273
 Linter, 273
 SCA (software composition analysis), 273
 secret scanning, 273
 static analysis, 273
 secret management tools, 275–277

- training, 267–269
 - individuals, 270
 - organizations, 269
- COEP (cross-origin embedder policy), 47
- commenting, 103–104
- communication security, 39–40
 - cryptography, 39–40
 - HTTPS, 39–40
 - teams, 330
- compatibility, backward compatibility, 16
- compliance, 12–14
- configuration, 99
- configuration issues, 225–227
- continuous learning, 327–328
- continuous testing, 312–313
- cookies, 50
- COOP (Cross-Origin Opener Policy), 48
- CORB (Cross-Origin Read Blocking), 49
- CORP (Cross-Origin Resource Policy), 48
- corrective controls, 111
- CORS (Cross-Origin Resource Sharing), 48–49, 138
- COTS (configurable off-the-shelf) software, 99
- coverage-based testing, 283
- COVID-19, planning for failure, 9
- CRA (Canadian Revenue Agency), 106
- credential stuffing attack, 38
- crossing trust boundaries, 229–230
- CRUD (create-read-update-delete), 7
- cryptography, 39–40, 75
 - cryptographic shredding, 37
 - hashing, 75
 - .NET, 197
 - nonces, 186
- CSF (Cybersecurity Framework), 15
- CSP (Content Security Policy) header, 188
- CSPM (cloud security posture management), 92
- CSRF (Cross Site Request Forgery), 138, 215
- CSS (Cascading Style Sheets), 148. *See also* HTML/CSS
 - best practices, 148–149
 - things to avoid, 149
- CVE (Common Vulnerability Enumerator), 212
- CWE (Common Weakness Enumerator), 212

D

- DAC (discretionary access control), 34
- DAST (dynamic application security testing tool), 132–133, 194, 215, 276–277, 282–283, 286–287, 296–297, 298
- Burp Suite Pro, 277
- continuous testing and, 312
- fuzzing, 287–288
- Zap (Zed Attack Proxy), 277

- data breaches, 40–42
 - assume breach concept, 9
- data minimization, 41
- data spills, 40–42
- databases, 56
 - cloud providers, 58
 - encryption in transit, 57
 - least privilege access, 57
 - masking data, 58
 - networks, 56
 - passwords, 57
 - queries, monitoring, 57
 - restoring, 57
 - rollbacks, 59
 - security requirements, 257
 - servers, 56
- DBA (database administrator), 7, 57
- DBO (database owner), 7
- DDD (domain-driven development), 78–79
- decryption, 75
- Defense in Depth, 10
- DEP (Data Execution Protection), 219
- dependencies
 - SCA (software composition analysis), 87
 - secure design, 86–87
 - updating, 274–275
 - verifying, 274–275
- deployment, 295
 - CI/CD system, 295
 - access control, 300
 - breaking the build, 297–298
 - DAST (dynamic automated scanners), 296–297, 298
 - dependencies, 300
 - fuzzing, 297
 - laC scanners, 299
 - linting, 299
 - package integrity, 300
 - performance testing, 297
 - pipeline security, 299–302
 - SaaS (Software as a Service) products, 301
 - SAST tool, first generation, 296, 298
 - SCA (software composition analysis), 299
 - secret scanning, 298
 - stress testing, 297
- depth-based testing, 283
- deserialization, 66–67
- design. *See also* secure design
 - threat modeling, 260–262
- design flaws, 212–213
- design paradigms, 83
- detective controls, 111
- DevOps, CI/CD and, 301–302
- DFD (data flow diagram), 261, 263–264
- documentation, 104–106

- DORA (Digital Operational Resilience Act), 13
- DOS (denial-of-service) attacks, 109
- DPAPI (Windows Data Protection API), 195
- DR (disaster recovery) plan, 9, 315–317
- DREAM, 260
- dynamic scanners, 276–277
- dynamic testing, 282–283
- dynamically typed languages, 77–78

E

- EFF (Electronic Frontier Foundation), 39
- elevated privileges, 111–112
- emulators, 134
- encoding, 25
 - output encoding, 26–29
- encryption, 75
 - homemade, 39–40
 - in transit, 57
- error handling, 50–52, 69–70
 - log injection, 71
 - rules, 70
- escaping, 25
- ethical hackers, 289
- EU Cybersecurity Act, 13
- execution, managed, 196
- expectations, 323–325
- exploits, workarounds, 91
- Express.js, 182–183

F

- fail closed, 12
 - password managers, 74–75
 - resource locking, 73–74
- FEDRAMP (Federal Risk and Authorization Management Program), 190
- file handling issues, 230
- file management, 59
 - FIM (file integrity monitoring), 60
- file uploads, 61
 - memory management, 63–64
 - overflow management, 64–66
 - source code, 62–63
- FIM (file integrity monitoring), 60, 102
- FISMA (Federal Information Security Modernization Act), 13, 14
- Flask, 207–210
- flow analysis, 230–231
- Fortify SAST, 271
- frameworks, 22–23, 181
 - Flask, 207–210
 - .NET (Core), 194–199
 - Ruby on Rails, 199–204
 - Spring, 204–207
 - Spring Boot, 204–207

- web frameworks (JavaScript)
 - Angular, 186–190
 - Express.js, 182–183
 - jQuery, 190–192
 - React.js, 184–186
 - Vue.js, 192–194
- fuzzing, 287–288
 - CI/CD, 297

G

- GDPR (General Data Protection Regulation of 2018), 13, 37, 41
- goal setting, 324
- good habits, 319–322
- GRC (Governance, Risk, and Compliance), 82–83
- group policies, 34
- GUI (graphical user interface), 7

H

- HAR (HTML archive), 298
- hashing, 75
- HIPAA (Health Insurance Portability Act), 13
- HITECH (Health Information Technology for Economic and Clinical Health Act), 13
- hostnames, validation, 100
- HTML (Hypertext Markup Language), 148. *See also* HTML/CSS
 - best practices, 148–149
 - HTML5, 149–151
 - things to avoid, 149
- HTML/CSS
 - best practices, 148–149
 - HTML5, 149–151
 - things to avoid, 149
- HTTPS, 39–40

I

- IaC scanners, 273, 299
- IAST (Interactive Application Security Testing), 15, 215, 288–289
- IDE
 - plugins
 - IaC scanner, 273
 - Linter, 273
 - SCA (software composition analysis), 273
 - secret scanning, 273
 - static analysis, 273
 - tooling, 93
- identity, 31–32
- identity issues, 223–224
- IDOR (insecure direct object reference), 215
- immutable builds, 107–109
- implementation issues, 225–227

- causes, 215
- preventing, 215–216
- risk and, 215
- important functions, 97
- InfoSec (information security), 56
- infrastructure testing, 281
- injection, 220–221
 - blind SQL injection, 222
 - causes, 221
 - preventing, 221–222
 - risk, 221
- input
 - encoding, 25
 - escaping, 25
 - issues, 222–223
 - causes, 223
 - preventing, 223
 - risk, 223
 - sanitizing, 25
- input validation, 23–26
- insecure deserialization, 66
- instrumentation, IAST and, 288–289
- integrity
 - checks, 107–109
 - definition, 6
 - software supply chain, 107–108
- interoperability, 313
- invalid page faults, 217
- IoT (Internet of Things)
 - best practices, 140–141
 - scoping and, 282
- IPS/IDS (intrusion prevention/detection systems), 102
- IR (incident response) team, 313
- ISO (International Organization for Standardization) standards, 13, 15

J

- Java
 - best practices, 160–164
 - serialization, 164
 - things to avoid, 164–165
- JavaScript, 143–144
 - best practices, 144–146
 - things to avoid, 146–148
- web frameworks
 - Angular, 186–190
 - Express.js, 182–183
 - jQuery, 190–192
 - React.js, 184–186
 - Vue.js, 192–194
- job changes, 331–335
 - transitioning to security team, 330–331

- jQuery, 190–192
- JSX syntax, 184

K

- KISS (Keep It Simple Silly), 18

L

- languages
 - C#, 166–170
 - C/C++, 175–178
 - dynamically typed, 77–78
 - HTML/CSS, 148–151
 - Java, 160–165
 - JavaScript, 143–147
 - memory safe, 79–80, 219
 - Node.js, 157–160
 - PHP, 170–175
 - Python, 152–153
 - SQL (Structured Query Language), 154–157
 - statically typed, 77–78
 - strongly typed, 76–77
 - TypeScript, 165–166
 - weakly typed, 77–78
- laws, 12–14
- least privilege, 6–8
 - database security, 57
 - PoLP (principle of least privilege), 6
- legacy systems, 306
- limit overrun race condition, 238
- Lint, 273
- linting, 299
- log injection, 71
- Log4j, 86
- logging, 72–73, 307
- logic flaws, 212–213
 - preventing, 214
 - risk and, 213–214
 - sources, 213

M

- MAC (mandatory access control), 33
- maintenance
 - alerting
 - logs, 307
 - metrics, 307
 - traces, 307
 - BCP (business continuity planning), 315–317
 - blocking, 308
 - API gateways, 310–311
 - CDNS, 309–310
 - data scientists and, 311–312
 - RASP, 310
 - virtual patching, 310
 - WAFs, 309

- DR (disaster recovery) plan, 315–317
 - monitoring
 - logs, 307
 - metrics, 307
 - security alerts, 307
 - traces, 307
 - observability, 307
 - logs, 307
 - metrics, 307
 - traces, 307
 - security incidents, 313–314
 - testing, continuous, 312–313
- managed execution, 196
- man-in-the-middle attacks, 100
- manipulator-in-the-middle attack, 100
- manual testing, 284
 - APIs and, 284
 - web browsers, 284
 - web proxy tools, 284
- masking data, 58
- memory issues
 - buffer overflows, 216–217
 - buffer overreads, 217
 - causes, 219
 - memory leaks, 218
 - page faults, invalid, 217
 - RCE (remote code execution), 219
 - risk, 219
 - use after free, 218
 - variables, uninitialized, 218
- memory leaks, 218
- memory management
 - overflows, 63–64
 - avoiding, 64–66
 - shell code, 64
- memory-safe languages, 79–80, 219
- mentors, 332–335
- metrics, 307
- MFA (multifactor authentication), 57
- mobile applications
 - best practices, 134–137
 - scoping and, 282
- monitoring, 72–73
 - logs, 307
 - metrics, 307
 - security alerts, 307
 - traces, 307

N

- .NET (Core), 194–199
 - output encoding, 28
- network identity issues, 223–224
- NIS Directive (Directive on Security of Network and Information Systems), 13

- NIST (National Institute of Standards and Technology), 190
 - special publications, 13
- Node.js
 - best practices, 157–159
 - security concerns, 159–160
- nonces, 186
- Notice of Security Breach Act, 13
- null pointers, dereferencing, 218
- NY DFS (New York Department of Financial Services) Cybersecurity Regulation, 13

O

- object handling issues
 - BOLA (broken object level access control), 234–235
 - deserialization, 234–235
- OOP (object-oriented programming)
 - abstraction, 234
 - encapsulation, 233
 - inheritance, 232
 - polymorphism, 233–234
- observability, 307
 - logs, 307
 - metrics, 307
 - traces, 307
- Okta, 32
- OOP (object-oriented programming)
 - abstraction, 234
 - DDD (domain-driven development), 78–79
 - encapsulation, 233
 - inheritance, 232
 - polymorphism, 233–234
- ORMs, 29–31
- output encoding, 26–27
 - HTML context, 28
 - JavaScript context, 28–29
 - OWASP XSS Prevention Cheat Sheet, 27–28
- OWASP (Open Web Application Security Project), 14
 - threat modeling definition, 260
- OWASP DSOMM (DevSecOps Maturity Model), 14
- OWASP Foundation Slack channel, 73
- OWASP SAMM (Software Assurance Maturity Model), 14
- OWASP Security Knowledge Framework, 14
- OWASP XSS Prevention Cheat Sheet, 27–28

P

- page faults, invalid, 217
- parameterized queries, 29–31
- Partnership Model, 22
- passkeys, 38

password management, 37–40, 74–75
 90-day rotation, 191–192
 passwordless systems, 38
 peppering, 39
 salting, 38–39
 secure defaults, 91
 PASTA, 260
 patching, 17–18
 paved roads, 8–9
 PCI DSS (Payment Card Industry Data Security Standard), 13
 penetration testing, 282–283
 pentester issues, 284–285
 peppering, 39
 permissions policy, 46
 PHP, 170–175
 output encoding, 28
 physical security controls, 110
 PIPEDA (Personal Information Protection and Electronic Documents Act), 41
 planning for failure, 9
 PoLP (principle of least privilege), 6
 preventative controls, 111
 privacy, 67
 Internet cache, 68
 PHI (Personal Health Information), 68
 PII (Personally Identifiable Information), 68
 planning for, 17
 privileges
 elevated, 111–112
 least privilege, 6–8
 product owner, 260
 professional mentors, 332–335
 programming frameworks. *See* frameworks
 projects
 activities outline, 253–254
 planning, 254–255
 schedule, 254–255
 security requirements, 255–256
 databases, 257
 web application APIs, 256–257
 web applications, 256
 PWAs (progressive web apps), 134
 Python
 best practices, 152–153
 things to avoid, 153

Q

quantum computing, 76
 queries, parameterized, 29–31
 QuietAI SAST, 271

R

race conditions, 218, 237–240
 RASP (Run-Time Application Self Protection)
 tools, 15, 310

RBAC (role-based access control), 33
 RBAC (rule-based access control), 34
 RCE (remote code execution), 64, 204–205, 219
 React.js, 184–186
 readable code, 93–96
 Redux JS, 184
 regulations, 12–14
 release, 295. *See also* deployment
 integrity protection, 302–303
 security approval, 303
 replay attacks, 228–229
 resource issues, 240–241
 resource locking, 73–74
 resources for chapters, 343–361
 responsibilities, 323
 reusable code, 100–101
 risk, logic flaws and, 213–214
 rollbacks, 59
 route guards, 189
 routers, 189
 Ruby on Rails, 199–204

S

SaaS (Software as a Service), CI/CD
 systems, 301
 salting, 38–39
 same-origin policy, 47
 SAMM (Software Assurance Maturity Model), 14
 sanitizing input, 25
 SAST (static application security testing), 92,
 215, 298
 CI/CD system, 296
 code review and, 271
 continuous testing and, 312
 SBOM (software bill of materials), 190
 SCA (software composition analysis), 87, 215,
 273, 299
 continuous testing, 312
 dependencies, 274–275
 scope creep, 22–23
 scoping, 282
 scripting, XSS (cross-site scripting), 24, 27
 SDLC (secure system development lifecycle),
 21–22, 82
 breaking the build and, 297–298
 secret management, 35–37, 92
 issues, 235–237
 tools, 275–276
 secret scanning, 298
 IDEs and, 273
 secure cookies, 50
 secure defaults, 8–9
 developers, 92–93
 passwords, 91
 users, 90–91

- secure design
 - architecture whiteboarding, 263
 - bugs, 82
 - concept examples, 84
 - concepts, 261–263
 - dependency, 86–87
 - design paradigms, 83
 - design review, 83
 - DFD (data flow diagram), 263–264
 - how much, 84–85
 - patterns, 261–263
 - supply chains, 87–89
 - threat modeling, 83, 85
 - design documentation, 261
 - DFD (data flow diagram), 261
 - four question frame, 260
 - user stories, 264–265
- secure guardrails, static analysis, 272
- security
 - assumptions, 3–4
 - usable, 12
- security activities outline, 253–254
 - planning, 254–255
 - scheduling, 254–255
- security alerts, 307. *See also* alerting
- security by obscurity, 11
- security champions, 328–329
- security controls, 110
 - administrative controls, 110
 - corrective controls, 111
 - detective controls, 111
 - physical controls, 110
 - preventative controls, 111
 - technical controls, 110
- security frameworks, 14
 - ASVS (Application Security Verification Standard), 14
 - OWASP (Open Web Application Security Project), 14
 - OWASP DSOMM (DevSecOps Maturity Model), 14
 - OWASP SAMM (Software Assurance Maturity Model), 14
 - OWASP Security Knowledge Framework, 14
 - SAMM (Software Assurance Maturity Model), 14
 - Security Knowledge Framework, 14
- security headers
 - COEP (cross-origin embedder policy), 47
 - content security policy header, 44
 - Content-Security-Policy header, 45
 - COOP (Cross-Origin Opener Policy), 48
 - CORB (Cross-Origin Read Blocking), 49
 - CORP (Cross-Origin Resource Policy), 48
 - CORS (Cross-Origin Resource Sharing), 48–49
 - CSP (content security policy), 44
 - Expect-CT, 46
 - HPKP (Public Key Pinning Extension), 46
 - HSTS (Strict-Transport-Security), 45
 - metadata request headers, 43
 - permissions policy, 46
 - Referrer-Policy, 46
 - same-origin policy, 47
 - trusted types, 44
 - X-Content-Type-Options, 45
 - X-Frame, 45
 - XSS, DOM-based, 44
 - X-XSS-Protection, 46
- security hygiene, 296
- security incidents, 313
 - IR (incident response) team, 313
 - prioritizing, 313
- Security Knowledge Framework, 14
- security maintenance, 112–113
- security realm, 185
- SEHOP (Structured Exception Handling Overwrite Protection), 219
- Semgrep Code SAST, 271
- sensitive data exposure, 40–42
- separation of duties, 108
- serialization, 66–67
- serverless computing best practices, 138–139
- shell code, 64
- shielding, 308
 - API gateways, 310–311
 - CDNS (Content Delivery Networks), 309–310
 - data scientists and, 311–312
 - RASP (runtime application self-protection), 310
 - virtual patching, 310
 - WAFs (web application firewalls), 309
- Shostack, Adam, 260
- SIEM (security information and event management) system, 72
- SME (subject matter expert), 79–80, 260
- Snyk Code SAST, 271
- SOC (Service Organization Control), 15
- SOP (Same-Origin Policy), 138
- source code, file uploads, 62–63
- SPAs (single-page applications), 192
- Spring, 204–207
- Spring Boot, 204–207
- SQL (Structured Query Language)
 - blind SQL injection, 222
 - hardening database and server, 154–155
 - inline, 30–31
 - language security tips, 154
 - security concerns, 156–157
- SQLi (SQL injection), 30
- SSO (single sign-on), 32
- SSSU (security-savvy software user), 264–265

- Stack-Smashing Protection, 136
- state, 69
- stateful systems, 69
- static analysis, 270
 - free tools, 271
 - IDE and, 273
 - secure guardrails, 272
- statically typed languages, 77–78
- stored procedures, 30–31
- STRIDE, 260
- strongly typed languages, 76–77
- supply chain security, 10–11
 - software supply chain security tool, 87
- supply chains, 87–89
- system calls, 102

T

- teams
 - communication, 330
 - log importance, 72–73
 - reusable code and, 100–101
 - technical debt, 113
 - transitioning to security team, 330–331
 - uniformity and, 71
- technical debt, 113–114
- technical security controls, 110
- testing
 - AppSec, 289–290
 - automated, 286–287
 - black box testing, 277
 - bounty hunters, 289
 - bugs
 - bounty programs, 289–290
 - retesting fixes, 18
 - Burp Suite Pro, 277
 - clear box, 270
 - continuous, 312–313
 - coverage, 280
 - versus* depth, 281
 - coverage-based, 283
 - DAST (dynamic application security testing tool), 276–277, 282–283, 286–287
 - depth *versus* coverage, 281
 - depth-based, 283
 - dynamic, 282–283
 - ethical hackers, 289
 - fuzzing, 287–288
 - hiring testers, 283, 285
 - IAST (Interactive Application Security Testing), 288–289
 - infrastructure, 281
 - lower-level environments, 281–282
 - manual, 284
 - APIs and, 284
 - web browsers, 284

- web proxy tools, 284
- penetration testing, 282–283
- production, 281–282
- reasons for, 280
- results, 290
 - actioning, 291–292
 - format, 291
 - scoping, 282
- secret scanning, 298
- static analysis, 270
 - first generation tools, 271
 - free tools, 271
 - second generation tools, 271
 - secure guardrails, 272
- timing, 280, 282–284
- VA (vulnerability assessment) scanner, 281
- white box, 270
- Zap (Zed Attack Proxy), 277
- TFS (team foundation server), 62–63
- The Shellcoder's Handbook*, 333
- threat modeling, 16–17, 83, 85
 - Adam Shostack's Four Question Frame for Threat Modeling, 260
 - Attack Trees, 260
 - DFD (data flow diagram), 261, 263–264
 - DREAM, 260
 - OWASP definition, 260
 - PASTA, 260
 - STRIDE, 260
 - Trike, 260
 - VAST, 260
- timing attacks, 237
- TLS (Transport Layer Security), 75
- traces, 307
- training for coding, 267–269
 - continuous learning, 327–328
 - individuals, 270
 - organizations, 269
- transaction fraud, 227–228
- Trike, 260
- trust
 - crossing trust boundaries, 229–230
 - implied, 4
- Trusted Types, 188–189
- type safety, 196
- TypeScript, 165–166

U

- undefined state, 241–242
- unit testing *versus* SAST, 273
- unknown state, 241–242
- untrusted systems, 96–97
- updates, 17–18
- usable security, 12
- use after free, 218

- user consent, 106–107
- user feedback, 12
- user stories, secure design and, 264–265
- UUID (universal unique identifier), 136
- UX (user experience), 26

V

- VA (vulnerability assessment) scanner, 281
- validation
 - assumptions, 4
 - hostnames, 100
 - input validation, 23–26
- variables
 - dangling pointers, 218
 - null pointers, dereferencing, 218
 - uninitialized, 218
 - wild pointers, 218
- VAST, 260
- Veracode SAST, 271
- virtual patching, 310
- VMs (virtual machines), 107
- Vue.js, 192–194
- vulnerabilities, 211
 - access issues, 225
 - authentication issues, 224–225
 - authorization issues, 225
 - code bugs, 215–216
 - compiler issues, 220–222
 - configuration issues, 225–227
 - CVE (Common Vulnerability Enumerator), 212
 - design flaws, 212–213
 - file-handling issues, 230–231
 - fraudulent transactions, 227–228
 - identity issues, 224–225
 - implementation errors, 215–216
 - implementation issues, 225–227
 - injection, 220–222
 - input issues, 222–223
 - interpréter issues, 220–222
 - invalid page faults, 217
 - logic flaws, 212–213

- memory issues, 216–220
- object handling issues, 232–235
- overflows, 216–220
- race conditions, 237–240
- replay attacks, 228–229
- resource issues, 240–241
- secrets management issues, 235–237
- timing issues, 237–240
- trust boundaries, crossing, 229–230
- unknown state, 241–242

W

- WAFs (web application firewalls), 309
- waterfall development, 107
- weakly typed languages, 77–78
- web application APIs security requirement
 - plan, 256–257
- web applications security requirement
 - plans, 256
- web frameworks (JavaScript)
 - Angular, 186–190
 - Express.js, 182–183
 - jQuery, 190–192
 - React.js, 184–186
 - Vue.js, 192–194
- WebSocket, best practices, 137–138
- white box testing, 270
- wild pointers, 218
- Windows Active Directory. *See* AD (Active Directory)
- workarounds, exploits, 91
- WSS (WebSocket Protocol), 138

X

- XSS (cross-site scripting), 24, 27
- XXE (XML external entities), 226

Z

- Zap (Zed Attack Proxy), 277
- Zero Trust, 9–10
- zoning, assumptions, 4