SOFTWARE SECURITY SERIES

Foreword by Gary McGraw

SECURE PROGRAMMING WITH STATIC ANALYSIS

Brian Chess Jacob West

Praise for *Secure Programming with Static Analysis*

"We designed Java so that it could be analyzed statically. This book shows you how to apply advanced static analysis techniques to create more secure, more reliable software."

—Bill Joy
Co-founder of Sun Microsystems, co-inventor of the Java programming language

"If you want to learn how promising new code-scanning tools can improve the security of your software, then this is the book for you. The first of its kind, *Secure Programming with Static Analysis* is well written and tells you what you need to know without getting too bogged down in details. This book sets the standard."

—David Wagner
Associate Professor, University of California, Berkeley

"Brian and Jacob can write about software security from the 'been there. done that.' perspective. Read what they've written - it's chock full of good advice."

—Marcus Ranum
Inventor of the firewall, Chief Scientist, Tenable Security

"Over the past few years, we've seen several books on software security hitting the bookstores, including my own. While they've all provided their own views of good software security practices, this book fills a void that none of the others have covered. The authors have done a magnificent job at describing in detail how to do static source code analysis using all the tools and technologies available today. Kudos for arming the developer with a clear understanding of the topic as well as a wealth of practical guidance on how to put that understanding into practice. It should be on the required reading list for anyone and everyone developing software today."

—Kenneth R. van Wyk
President and Principal Consultant, KRvW Associates, LLC.

"Software developers are the first and best line of defense for the security of their code. This book gives them the security development knowledge and the tools they need in order to eliminate vulnerabilities before they move into the final products that can be exploited."

—Howard A. Schmidt
Former White House Cyber Security Advisor

"Modern artifacts are built with computer assistance. You would never think to build bridges, tunnels, or airplanes without the most sophisticated, state of the art tools. And yet, for some reason, many programmers develop their software without the aid of the best static analysis tools. This is the primary reason that so many software systems are replete with bugs that could have been avoided. In this exceptional book, Brian Chess and Jacob West provide an invaluable resource to programmers. Armed with the hands-on instruction provided in *Secure Programming with Static Analysis*, developers will finally be in a position to fully utilize technological advances to produce better code. Reading this book is a prerequisite for any serious programming."

—Avi Rubin, Ph.D.
Professor of Computer Science, Johns Hopkins University
President and co-Founder, Independent Security Evaluators

"Once considered an optional afterthought, application security is now an absolute requirement. Bad guys will discover how to abuse your software in ways you've yet to imagine—costing your employer money and damaging its reputation. Brian Chess and Jacob West offer timely and salient guidance to design security and resiliency into your applications from the very beginning. Buy this book now and read it tonight."

—Steve Riley
Senior Security Strategist, Trustworthy Computing, Microsoft Corporation

"Full of useful code examples, this book provides the concrete, technical details you need to start writing secure software today. Security bugs can be difficult to find and fix, so Chess and West show us how to use static analysis tools to reliably find bugs and provide code examples demonstrating the best ways to fix them. *Secure Programming with Static Analysis* is an excellent book for any software engineer and the ideal code-oriented companion book for McGraw's process-oriented *Software Security* in a software security course."

—James Walden
Assistant Professor of Computer Science, Northern Kentucky University

"Brian and Jacob describe the root cause of many of today's most serious security issues from a unique perspective: static source code analysis.

Using lots of real-world source code examples combined with easy-to-understand theoretical analysis and assessment, this book is the best I've read that explains code vulnerabilities in such a simple yet practical way for software developers."

—Dr. Gang Cheng

"Based on their extensive experience in both the software industry and academic research, the authors illustrate sound software security practices with solid principles. This book distinguishes itself from its peers by advocating practical static analysis, which I believe will have a big impact on improving software security."

—Dr. Hao Chen
Assistant Professor of Computer Science, UC Davis

Secure Programming with Static Analysis

Secure Programming with Static Analysis

Brian Chess
Jacob West

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data:

Chess, Brian.
Secure programming with static analysis / Brian Chess.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-42477-8
1. Computer security. 2. Debugging in computer science. 3. Computer software—Quality control. I. Title.

QA76.9.A25C443 2007
005.8—dc22

2007010226

ISBN-10: 0-321-42477-8
ISBN-13: 978-0-321-42477-8
Text printed in the United States on recycled paper at R. R. Donnelley in Crawfordsville, Indiana.
Third printing, August 2010
This product is printed digitally on demand.

To Sally and Simon, with love.
—Brian

In memory of the best teacher I ever had, my Dad.
—Jacob

Contents

Part I: Software Security and Static Analysis 1

1 The Software Security Problem 3

1.1 Defensive Programming Is Not Enough 4
1.2 Security Features != Secure Features 6
1.3 The Quality Fallacy 9
1.4 Static Analysis in the Big Picture 11
1.5 Classifying Vulnerabilities 14
The Seven Pernicious Kingdoms 15
1.6 Summary 19

2 Introduction to Static Analysis 21

2.1 Capabilities and Limitations of Static Analysis 22
2.2 Solving Problems with Static Analysis 24
Type Checking 24
Style Checking 26
Program Understanding 27
Program Verification and Property Checking 28
Bug Finding 32
Security Review 33
2.3 A Little Theory, a Little Reality 35
Success Criteria 36
Analyzing the Source vs. Analyzing Compiled Code 42
Summary 45

3 Static Analysis as Part of the Code Review Process 47

3.1 Performing a Code Review 48
The Review Cycle 48
Steer Clear of the Exploitability Trap 54
3.2 Adding Security Review to an Existing Development Process 56
Adoption Anxiety 58
Start Small, Ratchet Up 62
3.3 Static Analysis Metrics 62
Summary 69

4 Static Analysis Internals 71

4.1 Building a Model 72
Lexical Analysis 72
Parsing 73
Abstract Syntax 74
Semantic Analysis 76
Tracking Control Flow 77
Tracking Dataflow 80
Taint Propagation 82
Pointer Aliasing 82
4.2 Analysis Algorithms 83
Checking Assertions 84
Naïve Local Analysis 85
Approaches to Local Analysis 89
Global Analysis 91
Research Tools 94
4.3 Rules 96
Rule Formats 97
Rules for Taint Propagation 101
Rules in Print 103
4.4 Reporting Results 105
Grouping and Sorting Results 106
Eliminating Unwanted Results 108
Explaining the Significance of the Results 109
Summary 113

Part II: Pervasive Problems 115

5 Handling Input 117

5.1 What to Validate 119
Validate All Input 120
Validate Input from All Sources 121
Establish Trust Boundaries 130
5.2 How to Validate 132
Use Strong Input Validation 133
Avoid Blacklisting 137
Don't Mistake Usability for Security 142
Reject Bad Data 143
Make Good Input Validation the Default 144
Check Input Length 153
Bound Numeric Input 157
5.3 Preventing Metacharacter Vulnerabilities 160
Use Parameterized Requests 161
Path Manipulation 167
Command Injection 168
Log Forging 169
Summary 172

6 Buffer Overflow 175

6.1 Introduction to Buffer Overflow 176
Exploiting Buffer Overflow Vulnerabilities 176
Buffer Allocation Strategies 179
Tracking Buffer Sizes 186
6.2 Strings 189
Inherently Dangerous Functions 189
Bounded String Operations 195
Common Pitfalls with Bounded Functions 203
Maintaining the Null Terminator 213
Character Sets, Representations, and Encodings 218
Format Strings 224
Better String Classes and Libraries 229
Summary 233

7 Bride of Buffer Overflow 235

7.1 Integers 236
Wrap-Around Errors 236
Truncation and Sign Extension 239
Conversion between Signed and Unsigned 241
Methods to Detect and Prevent Integer Overflow 242
7.2 Runtime Protection 251
Safer Programming Languages 251
Safer C Dialects 255
Dynamic Buffer Overflow Protections 258
Dynamic Protection Benchmark Results 263
Summary 263

8 Errors and Exceptions 265

8.1 Handling Errors with Return Codes 266
Checking Return Values in C 266
Checking Return Values in Java 269
8.2 Managing Exceptions 271
Catch Everything at the Top Level 272
The Vanishing Exception 273
Catch Only What You're Prepared to Consume 274
Keep Checked Exceptions in Check 276
8.3 Preventing Resource Leaks 278
C and C++ 279
Java 283
8.4 Logging and Debugging 286
Centralize Logging 286
Keep Debugging Aids and Back-Door Access Code out of Production 289
Clean Out Backup Files 292
Do Not Tolerate Easter Eggs 293
Summary 294

Part III: Features and Flavors 295

9 Web Applications 297

9.1 Input and Output Validation for the Web 298
Expect That the Browser Has Been Subverted 299
Assume That the Browser Is an Open Book 302
Protect the Browser from Malicious Content 303
9.2 HTTP Considerations 319
Use POST, Not GET 319
Request Ordering 322
Error Handling 322
Request Provenance 327
9.3 Maintaining Session State 328
Use Strong Session Identifiers 329
Enforce a Session Idle Timeout and a Maximum Session Lifetime 331
Begin a New Session upon Authentication 333
9.4 Using the Struts Framework for Input Validation 336
Setting Up the Struts Validator 338
Use the Struts Validator for All Actions 338
Validate Every Parameter 342
Maintain the Validation Logic 343
Summary 346

10 XML and Web Services 349

10.1 Working with XML 350
Use a Standards-Compliant XML Parser 350
Turn on Validation 352
Be Cautious about External References 358
Keep Control of Document Queries 362
10.2 Using Web Services 366
Input Validation 366
WSDL Worries 368
Over Exposure 369
New Opportunities for Old Errors 370
JavaScript Hijacking: A New Frontier 370
Summary 376

11 Privacy and Secrets 379

11.1 Privacy and Regulation 380
Identifying Private Information 380
Handling Private Information 383
11.2 Outbound Passwords 388
Keep Passwords out of Source Code 389
Don't Store Clear-Text Passwords 391
11.3 Random Numbers 397
Generating Random Numbers in Java 398
Generating Random Numbers in C and C++ 401
11.4 Cryptography 407
Choose a Good Algorithm 407
Don't Roll Your Own 409
11.5 Secrets in Memory 412
Minimize Time Spent Holding Secrets 414
Share Secrets Sparingly 415
Erase Secrets Securely 416
Prevent Unnecessary Duplication of Secrets 418
Summary 420

12 Privileged Programs 421

12.1 Implications of Privilege 423
Principle of Least Privilege 423
This Time We Mean It: Distrust Everything 426
12.2 Managing Privilege 427
Putting Least Privilege into Practice 427
Restrict Privilege on the Filesystem 433
Beware of Unexpected Events 436
12.3 Privilege Escalation Attacks 439
File Access Race Conditions 440
Insecure Temporary Files 446
Command Injection 450
Standard File Descriptors 452
Summary 454

Part IV: Static Analysis in Practice 457

13 Source Code Analysis Exercises for Java 459

Exercise 13.0 Installation 460
Exercise 13.1 Begin with the End in Mind 461
Exercise 13.2 Auditing Source Code Manually 469
Exercise 13.3 Running Fortify SCA 471
Exercise 13.4 Understanding Raw Analysis Results 472
Exercise 13.5 Analyzing a Full Application 478
Exercise 13.6 Tuning Results with Audit Workbench 479
Exercise 13.7 Auditing One Issue 483
Exercise 13.8 Performing a Complete Audit 487
Exercise 13.9 Writing Custom Rules 491
Answers to Questions in Exercise 13.2 499

14 Source Code Analysis Exercises for C 503

Exercise 14.0 Installation 504
Exercise 14.1 Begin with the End in Mind 505
Exercise 14.2 Auditing Source Code Manually 513
Exercise 14.3 Running Fortify SCA 514
Exercise 14.4 Understanding Raw Analysis Results 515
Exercise 14.5 Analyzing a Full Application 520
Exercise 14.6 Tuning Results with Audit Workbench 521
Exercise 14.7 Auditing One Issue 525
Exercise 14.8 Performing a Complete Audit 529
Exercise 14.9 Writing Custom Rules 531
Answers to Questions in Exercise 14.2 537

Epilogue 541

References 545

Index 559

This page intentionally left blank

Foreword

Software Security and Code Review with a Static Analysis Tool

On the first day of class, mechanical engineers learn a critical lesson: Pay attention and learn this stuff, or the bridge you build could fall down. This lesson is most powerfully illustrated by a video of the Tacoma Narrows Bridge shaking itself to death (http://www.enm.bris.ac.uk/anm/tacoma/tacoma.html). Figure 1 shows a 600-foot section of the bridge falling into the water in 1940. By contrast, on the first day of software engineering class, budding developers are taught that they can build anything that they can dream of. They usually start with "hello world."

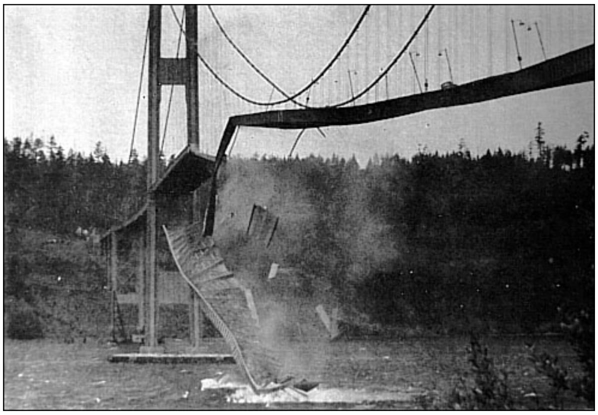
Figure 1 A 600-foot section of the Tacoma Narrows bridge crashes into Puget Sound as the bridge twists and torques itself to death. Mechanical engineers are warned early on that this can happen if they don't practice good engineering.

An overly optimistic approach to software development has certainly led to the creation of some mind-boggling stuff, but it has likewise allowed us to paint ourselves into the corner from a security perspective. Simply put, we neglected to think about what would happen to our software if it were intentionally and maliciously attacked.

Much of today's software is so fragile that it barely functions properly when its environment is pristine and predictable. If the environment in which our fragile software runs turns out to be pugnacious and pernicious (as much of the Internet environment turns out to be), software fails spectacularly, splashing into the metaphorical Puget Sound.

The biggest problem in computer security today is that most systems aren't constructed with security in mind. Reactive network technologies such as firewalls can help alleviate obvious script kiddie attacks on servers, but they do nothing to address the real security problem: bad software. If we want to solve the computer security problem, we need to do more to build secure software.

Software security is the practice of building software to be secure and function properly under malicious attack. This book is about one of software security's most important practices: code review with a static analysis tool.

As practitioners become aware of software security's importance, they are increasingly adopting and evolving a set of best practices to address the problem. Microsoft has carried out a noteworthy effort under its Trustworthy Computing Initiative. Many Cigital customers are in the midst of enterprise scale software security initiatives. Most approaches in practice today encompass training for developers, testers, and architects; analysis and auditing of software artifacts; and security engineering. There's no substitute for working software security as deeply into the development process as possible and taking advantage of the engineering lessons software practitioners have learned over the years.

In my book *Software Security*, I introduce a set of seven best practices called *touchpoints*. Putting software security into practice requires making some changes to the way most organizations build software. The good news is that these changes don't need to be fundamental, earth shattering, or cost-prohibitive. In fact, adopting a straightforward set of engineering best practices, designed in such a way that security can be interleaved into existing development processes, is often all it takes.

Figure 2 specifies the software security touchpoints and shows how software practitioners can apply them to the various software artifacts produced during software development. This means understanding how to work security engineering into requirements, architecture, design, coding, testing, validation, measurement, and maintenance.

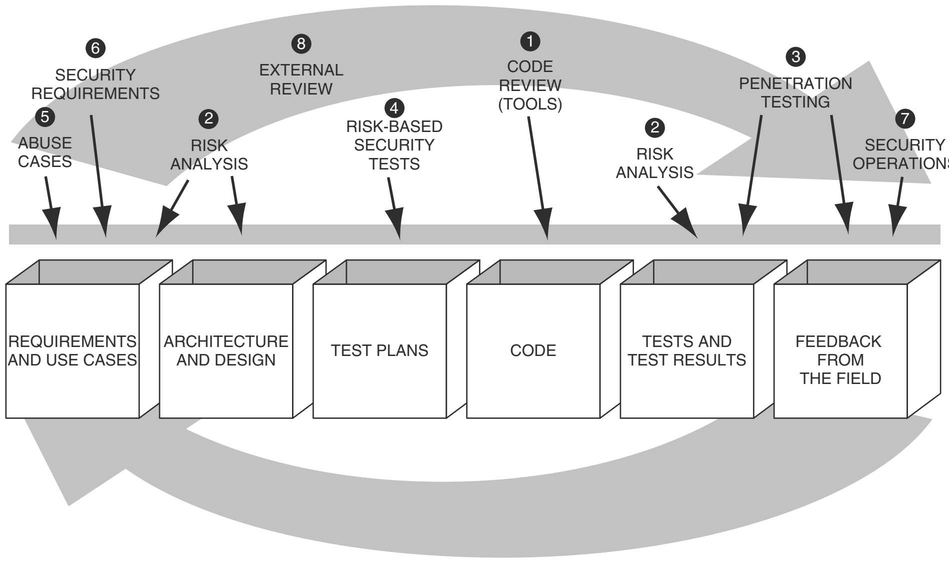

Figure 2 The software security touchpoints as introduced and fleshed out in *Software Security: Building Security In.*

Some touchpoints are, by their very nature, more powerful than others. Adopting the most powerful ones first is only prudent. The top two touchpoints are code review with a static analysis tool and architectural risk analysis. This book is all about the first.

All software projects produce at least one artifact: code. This fact moves code review to the number one slot on our list. At the code level, the focus is on implementation bugs, especially those that static analysis tools that scan source code for common vulnerabilities can discover. Several tools vendors now address this space, including Fortify Software, the company that Brian and Jacob work for.

Implementation bugs are both numerous and common (just like real bugs in the Virginia countryside), and include nasty creatures such as the notorious buffer overflow, which owes its existence to the use (or misuse) of vulnerable APIs (e.g., `gets()`, `strcpy()`, and so on in C). Code review processes, both manual and (even more important) automated with a static analysis tool, attempt to identify security bugs prior to the software's release.

Of course, no single technique is a silver bullet. Code review is a necessary but not sufficient practice for achieving secure software. Security bugs (especially in C and C++) are a real problem, but architectural flaws are just as big of a problem. Doing code review alone is an extremely useful activity, but given that this kind of review can only identify bugs, the best a code review can uncover is around 50% of the security problems. Architectural problems are very difficult (and mostly impossible) to find by staring at code. This is especially true for modern systems made of hundreds of thousands of lines of code. A comprehensive approach to software security involves holistically combining both code review and architectural analysis.

By its very nature, code review requires knowledge of code. An infosec practitioner with little experience writing and compiling software will be of little use during a code review. The code review step is best left in the hands of the members of the development organization, especially if they are armed with a modern source code analysis tool. With the exception of information security people who are highly experienced in programming languages and code-level vulnerability resolution, there is no natural fit for network security expertise during the code review phase. This might come as a great surprise to organizations currently attempting to impose software security on their enterprises through the infosec division. Even though the idea of security enforcement is solid, making enforcement at the code level successful when it comes to code review requires real hands-on experience with code.

The problem is that most developers have little idea what bugs to look for, or what to do about bugs if they do find them. That's where this book, *Secure Programming with Static Analysis*, comes in. The book that you have in your hands is the most advanced work on static analysis and code review for security ever released. It teaches you not only what the bugs are (what I sometimes call the "bug parade" approach to software security), but how to find them with modern static analysis tools and, more important, what to do to correct them. By putting the lessons in this book into practice, you go a long way toward helping to solve the software security problem.

Gary McGraw, Ph.D.
Berryville, Virginia
March 6, 2007

Company: www.cigital.com
Podcast: www.cigital.com/silverbullet
Blog: www.cigital.com/justiceleague
Book: www.swsec.com

Preface

Following the light of the sun, we left the Old World.
—Christopher Columbus

We live in a time of unprecedented economic growth, increasingly fueled by computer and communications technology. We use software to automate factories, streamline commerce, and put information into the hands of people who can act upon it. We live in the information age, and software is the primary means by which we tame information.

Without adequate security, we cannot realize the full potential of the digital age. But oddly enough, much of the activity that takes place under the guise of computer security isn't really about solving security problems at all; it's about cleaning up the mess that security problems create. Virus scanners, firewalls, patch management, and intrusion detection systems are all means by which we make up for shortcomings in software security. The software industry puts more effort into compensating for bad security than it puts into creating secure software in the first place. Do not take this to mean that we see no value in mechanisms that compensate for security failures. Just as every ship should have lifeboats, it is both good and healthy that our industry creates ways to quickly compensate for a newly discovered vulnerability. But the state of software security is poor. New vulnerabilities are discovered every day. In a sense, we've come to expect that we will need to use the lifeboats every time the ship sails.

Changing the state of software security requires changing the way software is built. This is not an easy task. After all, there are a limitless number of security mistakes that programmers could make! The potential for error might be limitless, but in practice, the programming community tends to repeat the same security mistakes. Almost two decades of buffer overflow vulnerabilities serve as an excellent illustration of this point. In 1988, the Morris worm made the Internet programming community aware that a buffer overflow could lead to a security breach, but as recently as 2004, buffer overflow was the number one cause of security problems cataloged by the Common Vulnerabilities and Exposures (CVE) Project [CWE, 2006]. This significant repetition of well-known mistakes suggests that many of the security problems we encounter today are preventable and that the software community possesses the experience necessary to avoid them.

We are thrilled to be building software at the beginning of the twenty-first century. It must have felt this way to be building ships during the age of exploration. When Columbus came to America, exploration was the driving force behind economic expansion, and ships were the means by which explorers traveled the world. In Columbus's day, being a world economic power required being a naval power because discovering a new land didn't pay off until ships could safely travel the new trade routes. Software security has a similar role to play in today's world. To make information technology pay off, people must trust the computer systems they use. Some pundits warn about an impending "cyber Armageddon," but we don't fear an electronic apocalypse nearly so much as we see software security as one of the primary factors that control the amount of trust people are willing to place in technology.

We believe that it is the responsibility of the people who create software to make sure that their creations are secure. Software security cannot be left to the system administrator or the end user. Network security, judicious administration, and wise use are all important, but in the long run, these endeavors cannot succeed if the software is inherently vulnerable. Although security can sometimes appear to be a black art or a matter of luck, we hope to show that it is neither. Making security sound impossible or mysterious is giving it more than its due. With the right knowledge and the right tools, good software security can be achieved by building security in to the software development process.

We sometimes encounter programmers who question whether software security is a worthy goal. After all, if no one hacked your software yesterday, why would you believe they'll hack it tomorrow? Security requires expending some extra thought, attention, and effort. This extra work wasn't nearly so important in previous decades, and programmers who haven't yet suffered security problems use their good fortune to justify continuing to ignore security. In his investigation of the loss of the space shuttle *Challenger,* Richard Feynman found that NASA had based its risk assessment on the fact that previous shuttle missions had been successful [Feynman, 1986]. They knew anomalous behavior had taken place in the past, but they used the fact that no disaster had occurred yet as a reason to believe that no disaster would ever occur. The resulting erosion of safety margins made failure almost inevitable. Feynman writes, "When playing Russian roulette, the fact that the first shot got off safely is little comfort for the next."

Secure Programming with Static Analysis

Two threads are woven throughout the book: software security and static source code analysis. We discuss a wide variety of common coding errors that lead to security problems, explain the security ramifications of each, and give advice for charting a safe course. Our most common piece of advice eventually found its way into the title of the book: Use static analysis tools to identify coding errors before they can be exploited. Our focus is on commercial software for both businesses and consumers, but our emphasis is on business systems. We won't get into the details that are critical for building software for purposes that imply special security needs. A lot could be said about the specific security requirements for building an operating system or an electronic voting machine, but we encounter many more programmers who need to know how to build a secure Web site or enterprise application.

Above all else, we hope to offer practical and immediately practicable advice for avoiding software security pitfalls. We use dozens of real-world examples of vulnerable code to illustrate the pitfalls we discuss, and the book includes a static source code analysis tool on a companion CD so that readers can experiment with the detection techniques we describe.

The book is not a guide to using security features, frameworks, or APIs. We do not discuss the Java Security Manager, advanced cryptographic techniques, or the right approach to identity management. Clearly, these are important topics. They are so important, in fact, that they warrant books of their own. Our goal is to focus on things unrelated to security features that put security at risk when they go wrong.

In many cases, the devil is in the details. Security principles (and violations of security principles) have to be mapped to their manifestation in source code. We've chosen to focus on programs written in C and Java because they are the languages we most frequently encounter today. We see plenty of other languages, too. Security-sensitive work is being done in C#, Visual Basic, PHP, Perl, Python, Ruby, and COBOL, but it would be difficult to write a single book that could even scratch the surface with all these languages.

In any case, many of the problems we discuss are language independent, and we hope that you will be able to look beyond the syntax of the examples to understand the ramifications for the languages you use.

Who Should Read the Book

This book is written for people who have decided to make software security a priority. We hope that programmers, managers, and software architects will all benefit from reading it. Although we do not assume any detailed knowledge about software security or static analysis, we cover the subject matter in enough depth that we hope professional code reviewers and penetration testers will benefit, too. We do assume that you are comfortable programming in either C or Java, and that you won't be too uncomfortable reading short examples in either language. Some chapters are slanted more toward one language than another. For instance, the examples in the chapters on buffer overflow are written in C.

How the Book Is Organized

The book is divided into four parts. Part I, "Software Security and Static Analysis," describes the big picture: the software security problem, the way static analysis can help, and options for integrating static analysis as part of the software development process. Part II, "Pervasive Problems," looks at pervasive security problems that impact software, regardless of its functionality, while Part III, "Features and Flavors," tackles security concerns that affect common varieties of programs and specific software features. Part IV, "Static Analysis in Practice," brings together Parts I, II, and III with a set of hands-on exercises that show how static analysis can improve software security.

Chapter 1, "The Software Security Problem," outlines the software security dilemma from a programmer's perspective: why security is easy to get wrong and why typical methods for catching bugs aren't very effective when it comes to finding security problems.

Chapter 2, "Introduction to Static Analysis," looks at the variety of problems that static analysis can solve, including structure, quality, and, of course, security. We take a quick tour of open source and commercial static analysis tools.

Chapter 3, "Static Analysis as Part of Code Review," looks at how static analysis tools can be put to work as part of a security review process. We examine the organizational decisions that are essential to making effective use of the tools. We also look at metrics based on static analysis output.

Chapter 4, "Static Analysis Internals," takes an in-depth look at how static analysis tools work. We explore the essential components involved in building a tool and consider the trade-offs that tools make to achieve good precision and still scale to analyze millions of lines of code.

Part II outlines security problems that are pervasive in software. Throughout the chapters in this section and the next, we give positive guidance for secure programming and then use specific code examples (many of them from real programs) to illustrate pitfalls to be avoided. Along the way, we point out places where static analysis can help.

Chapter 5, "Handling Input," addresses the most thorny software security topic that programmers have faced in the past, and the one they are most likely to face in the future: handling the many forms and flavors of untrustworthy input.

Chapter 6, "Buffer Overflow," and Chapter 7, "Bride of Buffer Overflow," look at a single input-driven software security problem that has been with us for decades: buffer overflow. Chapter 6 begins with a tactical approach: how to spot the specific code constructs that are most likely to lead to an exploitable buffer overflow. Chapter 7 examines indirect causes of buffer overflow, such as integer wrap-around. We then step back and take a more strategic look at buffer overflow and possible ways that the problem can be tamed.

Chapter 8, "Errors and Exceptions," addresses the way programmers think about unusual circumstances. Although errors and exceptions are only rarely the direct cause of security vulnerabilities, they are often related to vulnerabilities in an indirect manner. The connection between unexpected conditions and security problems is so strong that error handling and recovery will always be a security topic. At the end, the chapter discusses general approaches to logging and debugging, which is often integrally connected with error handling.

Part III uses the same style of positive guidance and specific code examples to tackle security concerns found in common types of programs and related to specific software features.

Chapter 9, "Web Applications," looks at the most popular security topic of the day: the World Wide Web. We look at security problems that are specific to the Web and to the HTTP protocol.

Chapter 10, “XML and Web Services,” examines a security challenge on the rise: the use of XML and Web Services to build applications out of distributed components.

Although security features are not our primary focus, some security features are so error prone that they deserve special treatment. Chapter 11, “Privacy and Secrets,” looks at programs that need to protect private information and, more generally, the need to maintain secrets. Chapter 12, “Privileged Programs,” looks at the special security requirements that must be taken into account when writing a program that operates with a different set of privileges than the user who invokes it.

Part IV is about gaining experience with static analysis. This book’s companion CD includes a static analysis tool, courtesy of our company, Fortify Software, and source code for a number of sample projects. Chapter 13, “Source Code Analysis Exercises for Java,” is a tutorial that covers static analysis from a Java perspective; Chapter 14, “Source Code Analysis Exercises for C,” does the same thing, but with examples and exercises written in C.

Conventions Used in the Book

Discussing security errors makes it easy to slip into a negative state of mind or to take a pessimistic outlook. We try to stay positive by focusing on what needs to be done to get security right. Specifics are important, though, so when we discuss programming errors, we try to give a working example that demonstrates the programming mistake under scrutiny. When the solution to a particular problem is far removed from our original example, we also include a rewritten version that corrects the problem. To keep the examples straight, we use an icon to denote code that intentionally contains a weakness:

⊗

We use a different icon to denote code where the weakness has been corrected:

✔

Other conventions used in the book include a monospaced font for code, both in the text and in examples.

Acknowledgments

Our editor at Addison-Wesley, Jessica Goldstein, has done more than just help us navigate the publishing process; a conversation with her at RSA 2005 got this project started. The rest of the crew at Addison-Wesley has been a great help (and very patient), too: Kristin Weinberger, Chris Zahn, Romny French, and Karen Gettman among others.

Portions of Chapters 1, 2, and 3 have their roots in technical papers and journal articles we've written in the last few years. We are grateful to our coauthors on those projects: Gary McGraw, Yekaterina Tsipenyuk O'Neil, Pravir Chandra, and John Steven.

Our reviewers suffered through some really rough rough drafts and always came back with constructive feedback. Many thanks to Gary McGraw, David Wagner, Geoff Morrison, Gary Hardy, Sean Fay, Richard Bejtlich, James Walden, Gang Cheng, Fredrick Lee, Steve Riley, and Hao Chen. We also received much-needed encouragement from Fortify's technical advisory board, including Gary McGraw, Marcus Ranum, Avi Rubin, Fred Schneider, Matt Bishop, Li Gong, David Wagner, Greg Morrisett, Bill Pugh, and Bill Joy.

Everyone at Fortify Software has been highly supportive of our work, and a significant amount of their work appears on the book's companion CD. We are enormously grateful for the support we've received. We also owe a huge debit of gratitude to Greg Nelson, who has shaped our views on static analysis.

Most of all, we give thanks to our families: Sally and Simon at Brian's house, and Jonathan at Jacob's house. It takes a lot of forbearance to live with someone who's working at a Silicon Valley software company, and putting up with someone who's writing software and writing a book at the same time is more than saintly. Finally, thanks to our parents. You set us down this road, and we wouldn't want to be headed anywhere else.

About the Authors

Brian Chess is a founder of Fortify Software. He currently serves as Fortify's Chief Scientist, where his work focuses on practical methods for creating secure systems. Brian holds a Ph.D. in Computer Engineering from the University of California at Santa Cruz, where he studied the application of static analysis to the problem of finding security-relevant defects in source code. Before settling on security, Brian spent a decade in Silicon Valley working at huge companies and small startups. He has done research on a broad set of topics, ranging from integrated circuit design all the way to delivering software as a service. He lives in Mountain View, California.

Jacob West manages Fortify Software's Security Research Group, which is responsible for building security knowledge into Fortify's products. Jacob brings expertise in numerous programming languages, frameworks, and styles together with knowledge about how real-world systems can fail. Before joining Fortify, Jacob worked with Professor David Wagner at the University of California at Berkeley to develop MOPS (MOdel Checking Programs for Security properties), a static analysis tool used to discover security vulnerabilities in C programs. When he is away from the keyboard, Jacob spends time speaking at conferences and working with customers to advance their understanding of software security. He lives in San Francisco, California.

3 Static Analysis as Part of the Code Review Process

In preparing for battle, plans are useless
but planning is indispensable.
—Dwight Eisenhower

There's a lot to know about how static analysis tools work. There's probably just as much to know about making static analysis tools work as part of a secure development process. In this respect, tools that assist with security review are fundamentally different than most other kinds of software development tools. A debugger, for example, doesn't require any organization-wide planning to be effective. An individual programmer can run it when it's needed, obtain results, and move on to another programming task. But the need for software security rarely creates the kind of urgency that leads a programmer to run a debugger. For this reason, an organization needs a plan for who will conduct security reviews, when the reviews will take place, and how to act on the results. Static analysis tools should be part of the plan because they can make the review process significantly more efficient.

Code review is a skill. In the first part of this chapter, we look at what that skill entails and outline the steps involved in performing a code review. We pay special attention to the most common snag that review teams get hung up on: debates about exploitability. In the second part of the chapter, we look at who needs to develop the code review skill and when they need to apply it. Finally, we look at metrics that can be derived from static analysis results.

3.1 Performing a Code Review

A security-focused code review happens for a number of different reasons:

- Some reviewers start out with the need to find a few exploitable vulnerabilities to prove that additional security investment is justified.
- For every large project that didn't begin with security in mind, the team eventually has to make an initial pass through the code to do a security retrofit.
- At least once in every release period, every project should receive a security review to account for new features and ongoing maintenance work.

Of the three, the second requires by far the largest amount of time and energy. Retrofitting a program that wasn't written to be secure can be a considerable amount of work. Subsequent reviews of the same piece of code will be easier. The initial review likely will turn up many problems that need to be addressed. Subsequent reviews should find fewer problems because programmers will be building on a stronger foundation.

Steve Lipner estimates that at Microsoft security activities consume roughly 20% of the release schedule the first time a product goes through Microsoft's Security Development Lifecycle. In subsequent iterations, security requires less than 10% of the schedule [Lipner, 2006]. Our experience with the code review phase of the security process is similar—after the backlog of security problems is cleared out, keeping pace with new development requires much less effort.

The Review Cycle

We begin with an overview of the code review cycle and then talk about each phase in detail. The four major phases in the cycle are:

1. Establish goals
2. Run the static analysis tool
3. Review code (using output from the tool)
4. Make fixes

Figure 3.1 shows a few potential back edges that make the cycle a little more complicated than a basic box step. The frequency with which the cycle is repeated depends largely upon the goals established in the first phase, but our experience is that if a first iteration identifies more than a handful of security problems, a second iteration likely will identify problems too.

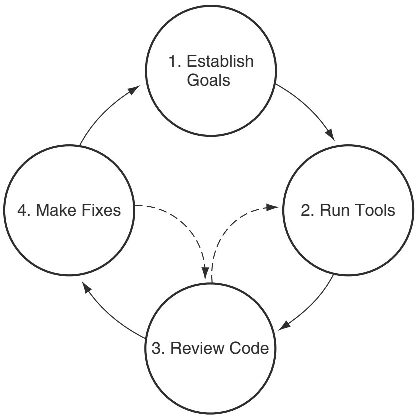

Figure 3.1 The code review cycle.

Later in the chapter, we discuss when to perform code review and who should do the reviewing, but we put forth a typical scenario here to set the stage. Imagine the first iteration of the cycle being carried out midway through the time period allocated for coding. Assume that the reviewers are programmers who have received security training.

1. Establish Goals

A well-defined set of security goals will help prioritize the code that should be reviewed and criteria that should be used to review it. Your goals should come from an assessment of the software risks you face. We sometimes hear sweeping high-level objectives along these lines:

- "If it can be reached from the Internet, it has to be reviewed before it's released."

  or

- "If it handles money, it has to be reviewed at least once a year."

We also talk to people who have more specific tactical objectives in mind. A short-term focus might come from a declaration:

- "We can't fail our next compliance audit. Make sure the auditor gives us a clean bill of health."

  or

- "We've been embarrassed by a series of cross-site scripting vulnerabilities. Make it stop."

You need to have enough high-level guidance to prioritize your potential code review targets. Set review priorities down to the level of individual programs. When you've gotten down to that granularity, don't subdivide any further; run static analysis on at least a whole program at a time. You might choose to review results in more detail or with greater frequency for parts of the program if you believe they pose more risk, but allow the tool's results to guide your attention, at least to some extent. At Fortify, we conduct line-by-line peer review for components that we deem to be high risk, but we always run tools against all of the code.

When we ask people what they're looking for when they do code review, the most common thing we hear is, "Uh, err, the OWASP Top Ten?" Bad answer. The biggest problem is the "?" at the end. If you're not too sure about what you're looking for, chances are good that you're not going to find it. The "OWASP Top Ten" part isn't so hot, either. Checking for the OWASP Top Ten is part of complying with the Payment Card Industry (PCI) Data Security Standard, but that doesn't make it the beginning and end of the kinds of problems you should be looking for. If you need inspiration, examine the results of previous code reviews for either the program you're planning to review or similar programs. Previously discovered errors have an uncanny way of slipping back in. Reviewing past results also gives you the opportunity to learn about what has changed since the previous review.

Make sure reviewers understand the purpose and function of the code being reviewed. A high-level description of the design helps a lot. It's also the right time to review the risk analysis results relevant to the code. If reviewers don't understand the risks before they begin, the relevant risks will inevitably be determined in an ad-hoc fashion as the review proceeds. The results will be less than ideal because the collective opinion about what is acceptable and what is unacceptable will evolve as the review progresses. The "I'll know a security problem when I see it" approach doesn't yield optimal results.

2. Run Static Analysis Tools

Run static analysis tools with the goals of the review in mind. To get started, you need to gather the target code, configure the tool to report the kinds of problems that pose the greatest risks, and disable checks that aren't relevant. The output from this phase will be a set of raw results for use during code review. Figure 3.2 illustrates the flow through phases 2 and 3.

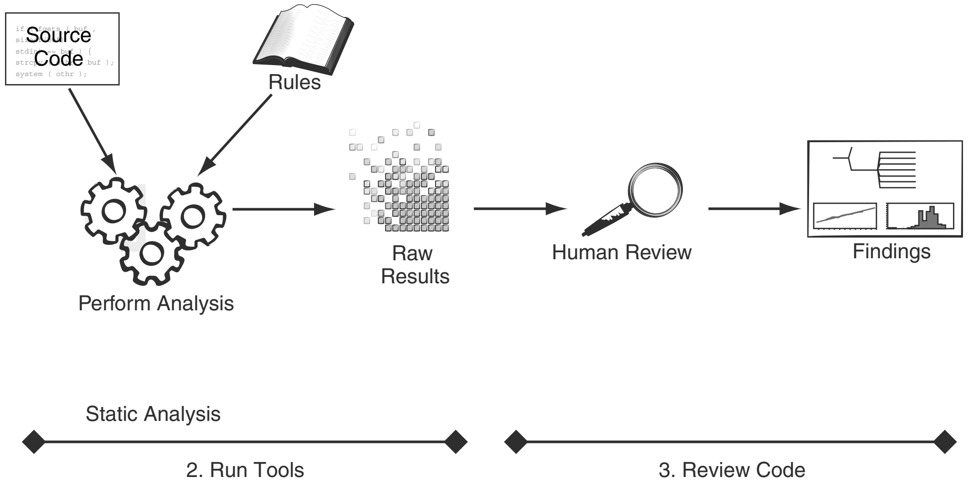

Figure 3.2 Steps 2 and 3: running the tool and reviewing the code.

To get good results, you should be able to compile the code being analyzed. For development groups operating in their own build environment, this is not much of an issue, but for security teams who've had the code thrown over the wall to them, it can be a really big deal. Where are all the header files? Which version of that library are you using? The list of snags and roadblocks can be lengthy. You might be tempted to take some shortcuts here. A static analysis tool can often produce at least some results even if the code doesn't compile. Don't cave. *Get the code into a compilable state before you analyze it.* If you get into the habit of ignoring parse errors and resolution warnings from the static analysis tool, you'll eventually miss out on important results.

This is also the right time to add custom rules to detect errors that are specific to the program being analyzed. If your organization has a set of secure coding guidelines, go through them and look for things you can encode as custom rules. A static analysis tool won't, by default, know what constitutes a security violation in the context of your code. Chances are good that you can dramatically improve the quality of the tool's results by customizing it for your environment.

Errors found during previous manual code reviews are particularly useful here, too. If a previously identified error can be phrased as a violation of some program invariant (never do X, or always do Y), write a rule to detect similar situations. Over time, this set of rules will serve as a form of institutional memory that prevents previous security slip-ups from being repeated.

3. Review Code

Now it's time to review the code with your own eyes. Go through the static analysis results, but don't limit yourself to just analysis results. Allow the tool to point out potential problems, but don't allow it to blind you to other problems that you can find through your own inspection of the code. We routinely find other bugs right next door to a tool-reported issue. This "neighborhood effect" results from the fact that static analysis tools often report a problem when they become confused in the vicinity of a sensitive operation. Code that is confusing to tools is often confusing to programmers, too, although not always for the same reasons. Go through all the static analysis results; don't stop with just the high-priority warnings. If the list is long, partition it so that multiple reviewers can share the work.

Reviewing a single issue is a matter of verifying the assumptions that the tool made when it reported the issue. Do mitigating factors prevent the code from being vulnerable? Is the source of untrusted data actually untrusted? Is the scenario hypothesized by the tool actually feasible?[1] If you are reviewing someone else's code, it might be impossible for you to answer all these questions, and you should collaborate with the author or owner of the code. Some static analysis tools make it easy to share results (for instance, by publishing an issue on an internal Web site), which simplifies this process.

Collaborative auditing is a form of peer review. Structured peer reviews are a proven technique for identifying all sorts of defects [Wiegers, 2002; Fagan, 1976]. For security-focused peer review, it's best to have a security specialist as part of the review team. Peer review and static analysis are complimentary techniques. When we perform peer reviews, we usually put one reviewer in charge of going through tool output.

If, during the review process, you identify a problem that wasn't found using static analysis, return to step 2: Write custom rules to detect other instances of the same problem and rerun the tools. Human eyes are great for spotting new varieties of defects, and static analysis excels at making sure that every instance of those new problems has been found. The back edge from step 3 to step 2 in Figure 3.1 represents this work.

1. Michael Howard outlines a structured process for answering questions such as these in a security and privacy article entitled "A Process for Performing Security Code Reviews" [Howard, 2006].

Code review results can take a number of forms: bugs entered into the bug database, a formal report suitable for consumption by both programmers and management, entries into a software security tracking system, or an informal task list for programmers. No matter what the form is, make sure the results have a permanent home so that they'll be useful during the next code review. Feedback about each issue should include a detailed explanation of the problem, an estimate of the risk it brings, and references to relevant portions of the security policy and risk assessment documents. This permanent collection of review results is good for another purpose, too: input for security training. You can use review results to focus training on real problems and topics that are most relevant to your code.

4. Make Fixes

Two factors control the way programmers respond to the feedback from a security review:

- Does security matter to them? If getting security right is a prerequisite for releasing their code, it matters. Anything less is shaky ground because it competes with adding new functionality, fixing bugs, and making the release date.
- Do they understand the feedback? Understanding security issues requires security training. It also requires the feedback to be written in an intelligible manner. Results stemming from code review are not concrete the way a failing test case is, so they require a more complete explanation of the risk involved.

If security review happens early enough in the development lifecycle, there will be time to respond to the feedback from the security review. Is there a large clump of issues around a particular module or a particular feature? It might be time to step back and look for design alternatives that could alleviate the problem. Alternatively, you might find that the best and most lasting fix comes in the form of additional security training.

When programmers have fixed the problems identified by the review, the fixes must be verified. The form that verification takes depends on the nature of the changes. If the risks involved are not small and the changes are nontrivial, return to the review phase and take another look at the code. The back edge from step 4 to step 3 in Figure 3.1 represents this work.

Steer Clear of the Exploitability Trap

Security review should not be about creating flashy exploits, but all too often, review teams get pulled down into exploit development. To understand why, consider the three possible verdicts that a piece of code might receive during a security review:

- Obviously exploitable
- Ambiguous
- Obviously secure

No clear dividing line exists between these cases; they form a spectrum. The endpoints on the spectrum are less trouble than the middle; obviously exploitable code needs to be fixed, and obviously secure code can be left alone. The middle case, ambiguous code, is the difficult one. Code might be ambiguous because its logic is hard to follow, because it's difficult to determine the cases in which the code will be called, or because it's hard to see how an attacker might be able to take advantage of the problem.

The danger lies in the way reviewers treat the ambiguous code. If the onus is on the reviewer to prove that a piece of code is exploitable before it will be fixed, the reviewer will eventually make a mistake and overlook an exploitable bug. When a programmer says, "I won't fix that unless you can prove it's exploitable," you're looking at the exploitability trap. (For more ways programmers try to squirm out of making security fixes, see the sidebar "Five Lame Excuses for Not Fixing Bad Code.")

The exploitability trap is dangerous for two reasons. First, developing exploits is time consuming. The time you put into developing an exploit would almost always be better spent looking for more problems. Second, developing exploits is a skill unto itself. What happens if you can't develop an exploit? Does it mean the defect is not exploitable, or that you simply don't know the right set of tricks for exploiting it?

Don't fall into the exploitability trap: Get the bugs fixed!

If a piece of code isn't obviously secure, make it obviously secure. Sometimes this approach leads to a redundant safety check. Sometimes it leads to a comment that provides a verifiable way to determine that the code is okay. And sometimes it plugs an exploitable hole. Programmers aren't always wild about the idea of changing a piece of code when no error can be demonstrated because any change brings with it the possibility of introducing a new bug. But the alternative—shipping vulnerabilities—is even less attractive.

Beyond the risk that an overlooked bug might eventually lead to a new exploit is the possibility that the bug might not even need to be exploitable to cause damage to a company's reputation. For example, a "security researcher" who finds a new buffer overflow might be able to garner fame and glory by publishing the details, even if it is not possible to build an attack around the bug [Wheeler, 2005]. Software companies sometimes find themselves issuing security patches even though all indications are that a defect isn't exploitable.

Five Lame Excuses for Not Fixing Bad Code

Programmers who haven't figured out software security come up with some inspired reasons for not fixing bugs found during security review. "I don't think that's exploitable" is the all-time winner. All the code reviewers we know have their own favorite runners-up, but here are our favorite specious arguments for ignoring security problems:

1. **"I trust system administrators."**

Even though I know they've misconfigured the software before, I know they're going to get it right this time, so I don't need code that verifies that my program is configured reasonably.

2. **"You have to authenticate before you can access that page."**

How on earth would an attacker ever get a username and a password? If you have a username and a password, you are, by definition, a good guy, so you won't attack the system.

3. **"No one would ever think to do that!"**

The user manual very clearly states that names can be no longer than 26 characters, and the GUI prevents you from entering any more than 26 characters. Why would I need to perform a bounds check when I read a saved file?

4. **"That function call can never fail."**

I've run it a million times on my Windows desktop. Why would it fail when it runs on the 128 processor Sun server?

5. **"We didn't intend for that to be production-ready code."**

Yes, we know it's been part of the shipping product for several years now, but when it was written, we didn't expect it to be production ready, so you should review it with that in mind.

3.2 Adding Security Review to an Existing Development Process[2]

It's easy to talk about integrating security into the software development process, but it can be a tough transition to make if programmers are in the habit of ignoring security. Evaluating and selecting a static analysis tool can be the easiest part of a software security initiative. Tools can make programmers more efficient at tackling the software security problem, but tools alone cannot solve the problem. In other words, static analysis should be used as part of a secure development lifecycle, not as a replacement for a secure development lifecycle.

Any successful security initiative requires that programmers buy into the idea that security is important. In traditional hierarchical organizations, that usually means a dictum from management on the importance of security, followed by one or more signals from management that security really should be taken seriously. The famous 2002 memo from Bill Gates titled "Trustworthy Computing" is a perfect example of the former. In the memo, Gates wrote:

> So now, when we face a choice between adding features and resolving security issues, we need to choose security.

Microsoft signaled that it really was serious about security when it called a halt to Windows development in 2002 and had the entire Windows division (upward of 8,000 engineers) participate in a security push that lasted for more than two months [Howard and Lipner, 2006].

Increasingly, the arrival of a static analysis tool is part of a security push. For that reason, adoption of static analysis and adoption of an improved process for security are often intertwined. In this section, we address the hurdles related to tool adoption. Before you dive in, read the adoption success stories in the sidebar "Security Review Times Two."

Security Review Times Two

Static analysis security tools are new enough that, to our knowledge, no formal studies have been done to measure their impact on the software built by large organizations. But as part of our work at Fortify, we've watched closely as our customers have rolled out our tools to their development teams and security organizations. Here we describe

2. This section began as an article in *IEEE Security & Privacy Magazine*, co-authored with Pravir Chandra and John Steven [Chandra, Chess, Steven, 2006].

the results we’ve seen at two large financial services companies. Because the companies don't want their names to be used, we'll call them “East Coast” and “West Coast.”

East Coast

A central security team is charged with doing code review. Before adopting a tool, the team reviewed 10 million lines of code per year. With Fortify, they are now reviewing 20 million lines of code per year. As they have gained familiarity with static analysis, they have written custom rules to enforce larger portions of their security policy. The result is that, as the tools do more of the review work, the human reviewers continue to become more efficient. In the coming year, they plan to increase the rate of review to 30 million lines of code per year without growing the size of the security team.

Development groups at the company are starting to adopt the tool, too; more than 100 programmers use the tool as part of the development process, but the organization has not yet measured the impact of developer adoption on the review process.

West Coast

A central security team is charged with reviewing all Internet-facing applications before they go to production. In the past, it took the security team three to four weeks to perform a review. Using static analysis, the security team now conducts reviews in one to two weeks. The security team expects to further reduce the review cycle time by implementing a process wherein the development team can run the tool and submit the results to the security team. (This requires implementing safeguards to ensure that the development team runs the analysis correctly.) The target is to perform code review for most projects in one week.

The security team is confident that, with the addition of source code analysis to the review process, they are now finding 100% of the issues in the categories they deem critical (such as cross-site scripting). The previous manual inspection process did not allow them to review every line of code, leaving open the possibility that some critical defects were being overlooked.

Development teams are also using static analysis to perform periodic checks before submitting their code to the security team. Several hundred programmers have been equipped with the tool. The result is that the security team now finds critical defects only rarely. (In the past, finding critical defects was the norm.) This has reduced the number of schedule slips and the number of “risk-managed deployments” in which the organization is forced to field an application with known vulnerabilities. The reduction in critical defects also significantly improves policy enforcement because when a security problem does surface, it now receives appropriate attention.

As a side benefit, development teams report that they routinely find non-security defects as a result of their code review efforts.

Adoption Anxiety

All the software development organizations we've ever seen are at least a little bit chaotic, and changing the behavior of a chaotic system is no mean feat. At first blush, adopting a static analysis tool might not seem like much of a problem. Get the tool, run the tool, fix the problems, and you're done. Right? Wrong. It's unrealistic to expect attitudes about security to change just because you drop off a new tool. Adoption is not as easy as leaving a screaming baby on the doorstep. Dropping off the tool and waving goodbye will lead to objections like the ones in Table 3.1.

Table 3.1 Commonly voiced objections to static analysis and their true meaning.

Objection	Translation
"It takes too long to run."	"I think security is optional, and since it requires effort, I don't want to do it."
"It has too many false positives."	"I think security is optional, and since it requires effort, I don't want to do it."
"It doesn't fit in to the way I work."	"I think security is optional, and since it requires effort, I don't want to do it."

In our experience, three big questions must be answered to adopt a tool successfully. An organization's size, along with the style and maturity of its development processes, all play heavily into the answers to these questions. None of them has a one-size-fits-all answer, so we consider the range of likely answers to each. The three questions are:

- Who runs the tool?
- When is the tool run?
- What happens to the results?

Who Runs the Tool?

Ideally, it wouldn't matter who actually runs the tool, but a number of practical considerations make it an important question, such as access to the code. Many organizations have two obvious choices: the security team or the programmers.

The Security Team
For this to work, you must ensure that your security team has the right skill set—in short, you want security folks with software development chops. Even if you plan to target programmers as the main consumers of the information generated by the tool, having the security team participate is a huge asset. The team brings risk management experience to the table and can often look at big-picture security concerns, too. But the security team didn't write the code, so team members won't have as much insight into it as the developers who did. It's tough for the security team to go through the code alone. In fact, it can be tricky to even get the security team set up so that they can compile the code. (If the security team isn't comfortable compiling other people's code, you're barking up the wrong tree.) It helps if you already have a process in place for the security team to give code-level feedback to programmers.

The Programmers
Programmers possess the best knowledge about how their code works. Combine this with the vulnerability details provided by a tool, and you've got a good reason to allow development to run the operation. On the flip side, programmers are always under pressure to build a product on a deadline. It's also likely that, even with training, they won't have the same level of security knowledge or expertise as members of the security team. If the programmers will run the tool, make sure they have time built into their schedule for it, and make sure they have been through enough security training that they'll be effective at the job. In our experience, not all programmers will become tool jockeys. Designate a senior member of each team to be responsible for running the tool, making sure the results are used appropriately, and answering tool-related questions from the rest of the team.

All of the Above
A third option is to have programmers run the tools in a mode that produces only high-confidence results, and use the security team to conduct more thorough but less frequent reviews. This imposes less of a burden on the programmers, while still allowing them to catch some of their own mistakes. It also encourages interaction between the security team and the development team. No question about it, joint teams work best. Every so

often, buy some pizzas and have the development team and the security team sit down and run the tool together. Call it eXtreme Security, if you like.

When Is the Tool Run?

More than anything else, deciding when the tool will be run determines the way the organization approaches security review. Many possible answers exist, but the three we see most often are these: while the code is being written, at build time, and at major milestones. The right answer depends on how the analysis results will be consumed and how much time it takes to run the tool.

While the Code Is Being Written

Studies too numerous to mention have shown that the cost of fixing a bug increases over time, so it makes sense to check new code promptly. One way to accomplish this is to integrate the source code analysis tool into the programmer's development environment so that the programmer can run on-demand analysis and gain expertise with the tool over time. An alternate method is to integrate scanning into the code check-in process, thereby centralizing control of the analysis. (This approach costs the programmers in terms of analysis freedom, but it's useful when desktop integration isn't feasible.) If programmers will run the tool a lot, the tool needs to be fast and easy to use. For large projects, that might mean asking each developer to analyze only his or her portion of the code and then running an analysis of the full program at build time or at major milestones.

At Build Time

For most organizations, software projects have a well-defined build process, usually with regularly scheduled builds. Performing analysis at build time gives code reviewers a reliable report to use for direct remediation, as well as a baseline for further manual code inspection. Also, by using builds as a timeline for source analysis, you create a recurring, consistent measure of the entire project, which provides perfect input for analysis-driven metrics. This is a great way to get information to feed a training program.

At Major Milestones

Organizations that rely on heavier-weight processes have checkpoints at project milestones, generally near the end of a development cycle or at some large interval during development. These checkpoints sometimes include security-related tasks such as a design review or a penetration test. Logically extending this concept, checkpoints seem like a natural place to use a static analysis tool. The down side to this approach is that programmers might put off thinking about security until the milestone is upon them, at which point other milestone obligations can push security off to the sidelines. If you're going to wait for milestones to use static analysis, make sure you build some teeth into the process. The consequences for ignoring security need to be immediately obvious and known to all ahead of time.

What Happens to the Results?

When people think through the tool adoption process, they sometimes forget that most of the work comes after the tool is run. It's important to decide ahead of time how the actual code review will be performed.

Output Feeds a Release Gate

The security team processes and prioritizes the tool's output as part of a checkpoint at a project milestone. The development team receives the prioritized results along with the security team's recommendations about what needs to be fixed. The development team then makes decisions about which problems to fix and which to classify as "accepted risks." (Development teams sometimes use the results from a penetration test the same way.) The security team should review the development team's decisions and escalate cases where it appears that the development team is taking on more risk than it should. If this type of review can block a project from reaching a milestone, the release gate has real teeth. If programmers can simply ignore the results, they will have no motivation to make changes.

The gate model is a weak approach to security for the same reason that penetration testing is a weak approach to security: It's reactive. Even though the release gate is not a good long-term solution, it can be an effective stepping stone. The hope is that the programmers will eventually get tired of having their releases waylaid by the security team and decide to take a more proactive approach.

A Central Authority Doles Out Individual Results

A core group of tool users can look at the reported problems for one or more projects and pick the individual issues to send to the programmers responsible for the code in question. In such cases, the static analysis tools should report everything it can; the objective is to leave no stone unturned.

False positives are less of a concern because a skilled analyst processes the results prior to the final report. With this model, the core group of tool users becomes skilled with the tools in short order and becomes adept at going through large numbers of results.

A Central Authority Sets Pinpoint Focus
Because of the large number of projects that might exist in an organization, a central distribution approach to results management can become constrained by the number of people reviewing results, even if reviewers are quite efficient. However, it is not unusual for a large fraction of the acute security pain to be clustered tightly around just a small number of types of issues. With this scenario, the project team will limit the tool to a small number of specific problem types, which can grow or change over time according to the risks the organization faces. Ultimately, defining a set of in-scope problem types works well as a centrally managed policy, standard, or set of guidelines. It should change only as fast as the development team can adapt and account for all the problems already in scope. On the whole, this approach gives people the opportunity to become experts incrementally through hands-on experience with the tool over time.

Start Small, Ratchet Up

Security tools tend to come preconfigured to detect as much as they possibly can. This is really good if you're trying to figure out what a tool is capable of detecting, but it can be overwhelming if you're assigned the task of going through every issue. No matter how you answer the adoption questions, our advice here is the same: Start small. Turn off most of the things the tool detects and concentrate on a narrow range of important and well-understood problems. Broaden out only when there's a process in place for using the tool and the initial batch of problems is under control. No matter what you do, a large body of existing code won't become perfect overnight. The people in your organization will thank you for helping them make some prioritization decisions.

3.3 Static Analysis Metrics

Metrics derived from static analysis results are useful for prioritizing remediation efforts, allocating resources among multiple projects, and getting feedback on the effectiveness of the security process. Ideally, one could use metrics derived from static analysis results to help quantify the amount of risk associated with a piece of code, but using tools to measure risk is tricky. The most obvious problem is the unshakable presence of false positives and false negatives, but it is possible to compensate for them. By manually auditing enough results, a security team can predict the rate at which false positives and false negatives occur for a given project and extrapolate the number of true positives from a set of raw results. A deeper problem with using static analysis to quantify risk is that there is no good way to sum up the risk posed by a set of vulnerabilities. Are two buffer overflows twice as risky as a single buffer overflow? What about ten? Code-level vulnerabilities identified by tools simply do not sum into an accurate portrayal of risk. See the sidebar "The Density Deception" to understand why.

Instead of trying to use static analysis output to directly quantify risk, use it as a tactical way to focus security efforts and as an indirect measure of the process used to create the code.

The Density Deception

In the quality assurance realm, it's normal to compute the *defect density* for a piece of code by dividing the number of known bugs by the number of lines of code. Defect density is often used as a measure of quality. It might seem intuitive that one could use static analysis output to compute a "vulnerability density" to measure the amount of risk posed by the code. It doesn't work. We use two short example programs with some blatant vulnerabilities to explain why. First up is a straight-line program:

```
/* This program computes Body Mass Index (BMI). */
int main(int argc, char** argv)
{
  char heightString[12];
  char weightString[12];
  int height, weight;
  float bmi;

  printf("Enter your height in inches: ");
  gets(heightString);
  printf("Enter your weight in pounds: ");
  gets(weightString);
  height = atoi(heightString);
  weight = atoi(weightString);
  bmi = ((float)weight/((float)height*height)) * 703.0;

  printf("\nBody mass index is %2.2f\n\n", bmi);
}
```

Continues

Continued

The program has 18 lines, and any static analysis tool will point out two glaring buffer overflow vulnerabilities: the calls to `gets()` on lines 10 and 12. Divide 2 by 18 for a vulnerability density of 0.111. Now consider another program that performs exactly the same computation:

```
 1 /* This program computes Body Mass Index (BMI). */
 2 int main(int argc, char** argv)
 3 {
 4   int height, weight;
 5   float bmi;
 6
 7   height = getNumber("Enter your height in inches");
 8   weight = getNumber("Enter your weight in pounds");
 9   bmi = ((float)weight/((float)height*height)) * 703.0;
10
11   printf("\nBody mass index is %2.2f\n\n", bmi);
12 }
13
14 int getNumber(char* prompt) {
15   char buf[12];
16   printf("%s: ", prompt);
17   return atoi(gets(buf));
18 }
```

This program calls `gets()`, too, but it uses a separate function to do it. The result is that a static analysis tool will report only one vulnerability (the call to `gets()` on line 17). Divide 1 by 18 for a vulnerability density of 0.056. Whoa. The second program is just as vulnerable as the first, but its vulnerability density is 50% smaller! The moral to the story is that the way the program is written has a big impact on the vulnerability density. This makes vulnerability density completely meaningless when it comes to quantifying risk. (Stay tuned. Even though vulnerability density is terrible in this context, the next section describes a legitimate use for it.)

Metrics for Tactical Focus

Many simple metrics can be derived from static analysis results. Here we look at the following:

- Measuring vulnerability density
- Comparing projects by severity
- Breaking down results by category
- Monitoring trends

Measuring Vulnerability Density

We've already thrown vulnerability density under the bus, so what more is there to talk about? Dividing the number of static analysis results by the number of lines of code is an awful way to measure risk, but it's a good way to measure the amount of work required to do a complete review. Comparing vulnerability density across different modules or different projects helps formulate review priorities. Track issue density over time to gain insight into whether tool output is being taken into consideration.

Comparing Projects by Severity

Static analysis results can be applied for project comparison purposes. Figure 3.3 shows a comparison between two modules, with the source code analysis results grouped by severity. The graph suggests a plan of action: Check out the critical issues for the first module, and then move on to the high-severity issues for the second.

Comparing projects side by side can help people understand how much work they have in front of them and how they compare to their peers. When you present project comparisons, name names. Point fingers. Sometimes programmers need a little help accepting responsibility for their code. Help them.

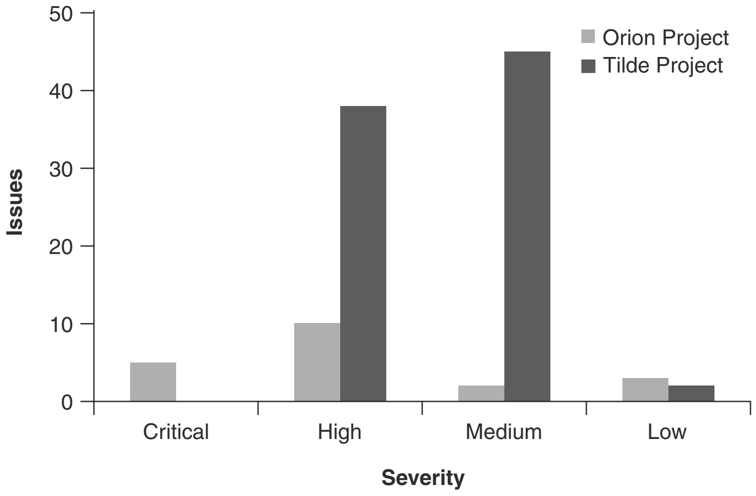

Figure 3.3 Source code analysis results broken down by severity for two subprojects.

Breaking Down Results by Category

Figure 3.4 presents results for a single project grouped by category. The pie chart gives a rough idea about the amount of remediation effort required to address each type of issue. It also suggests that log forging and cross-site scripting are good topics for an upcoming training class.

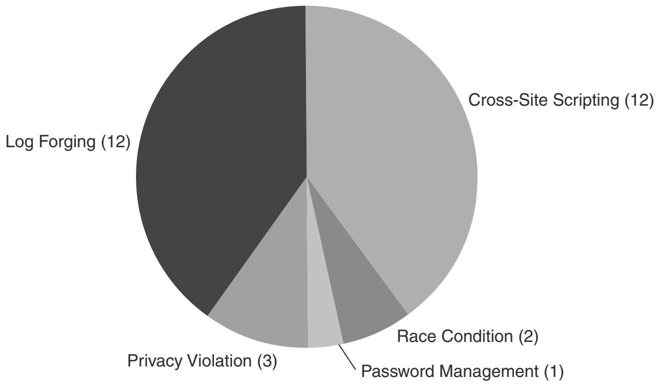

Figure 3.4 Source code analysis results broken down by category.

Source code analysis results can also point out trends over time. Teams that are focused on security will decrease the number of static analysis findings in their code. A sharp increase in the number of issues found deserves attention. Figure 3.5 shows the number of issues found during a series of nightly builds. For this particular project, the number of issues found on February 2 spikes because the development group has just taken over a module from a group that has not been focused on security.

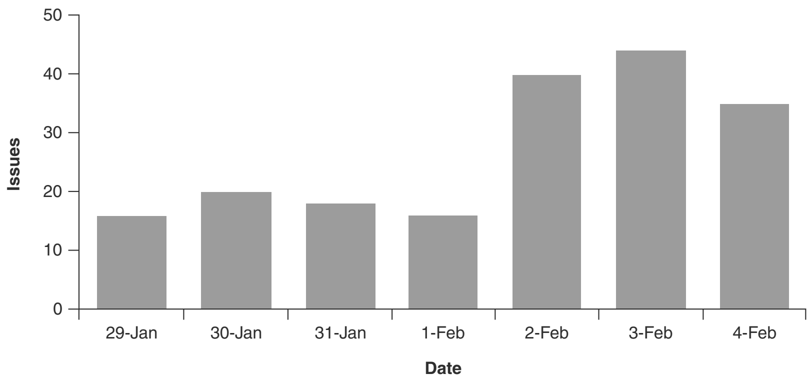

Figure 3.5 Source code analysis results from a series of nightly builds. The spike in issues on February 2 reflects the incorporation of a module originally written by a different team.

Process Metrics

The very presence of some types of issues can serve as an early indicator of more widespread security shortcomings [Epstein, 2006]. Determining the kinds of issues that serve as bellwether indicators requires some experience with the particular kind of software being examined. In our experience, a large number of string-related buffer overflow issues is a sign of trouble for programs written in C.

More sophisticated metrics leverage the capacity of the source code analyzer to give the same issue the same identifier across different builds. (See Chapter 4, "Static Analysis Internals," for more information on issue identifiers.) By following the same issue over time and associating it with the feedback provided by a human auditor, the source code analyzer can provide insight into the evolution of the project. For example, static analysis results can reveal the way a development team responds to security vulnerabilities. After an auditor identifies a vulnerability, how long, on average, does it take for the programmers to make a fix? We call this *vulnerability dwell*. Figure 3.6 shows a project in which the programmers fix critical vulnerabilities within two days and take progressively longer to address less severe problems.

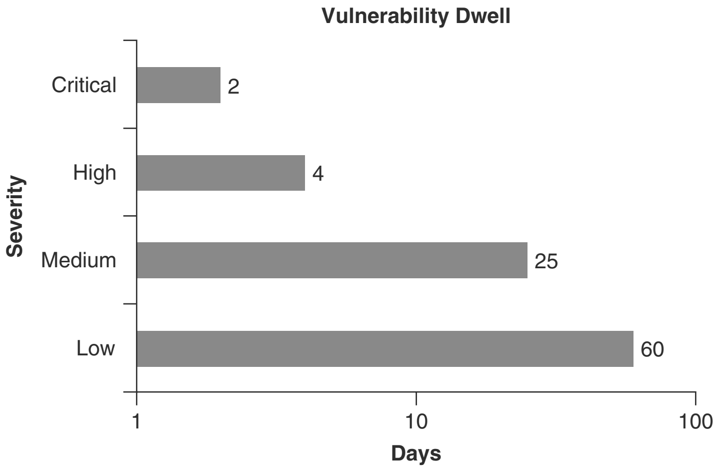

Figure 3.6 Vulnerability dwell as a function of severity. When a vulnerability is identified, vulnerability dwell measures how long it remains in the code. (The x-axis uses a log scale.)

Static analysis results can also help a security team decide when it's time to audit a piece of code. The rate of auditing should keep pace with the rate of development. Better yet, it should keep pace with the rate at which potential security issues are introduced into the code. By tracking individual issues over time, static analysis results can show a security team how many unreviewed issues a project contains. Figure 3.7 presents a typical graph. At the point the project is first reviewed, audit coverage goes to 100%. Then, as the code continues to evolve, the audit coverage decays until the project is audited again.

Another view of this same data gives a more comprehensive view of the project. An audit history shows the total number of results, number of results reviewed, and number of vulnerabilities identified in each build. This view takes into account not just the work of the code reviewers, but the effect the programmers have on the project. Figure 3.8 shows results over roughly one month of nightly builds. At the same time the code review is taking place, development is in full swing, so the issues in the code continue to change. As the auditors work, they report vulnerabilities (shown in black).

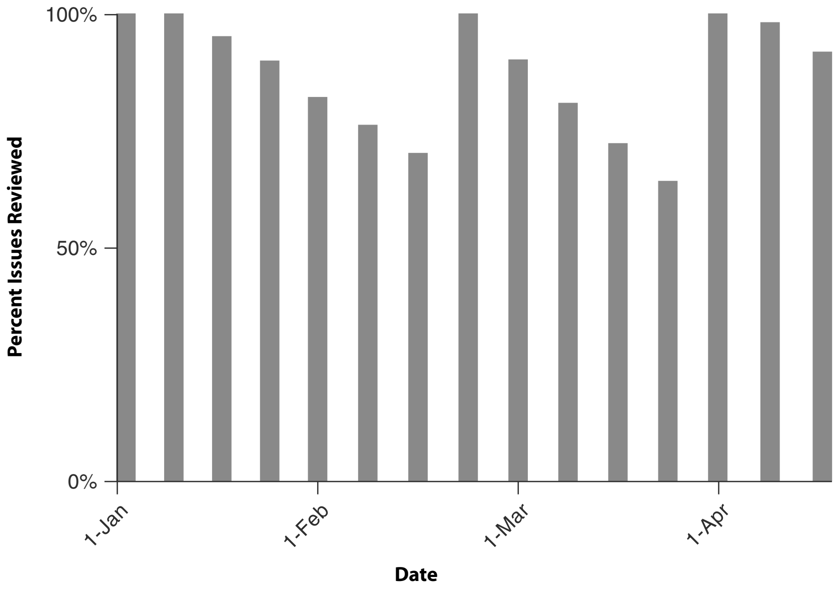

Figure 3.7 Audit coverage over time. After all static analysis results are reviewed, the code continues to evolve and the percentage of reviewed issues begins to decline.

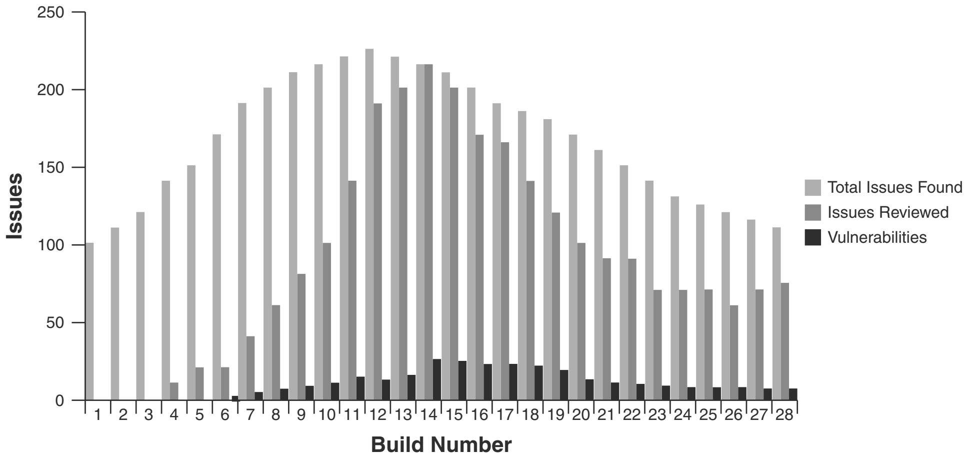

Figure 3.8 Audit history: the total number of static analysis results, the number of reviewed results, and the number of identified vulnerabilities present in the project.

Around build 14, the auditors have looked at all the results, so the total number of results is the same as the number reviewed. Development work is not yet complete, though, and soon the project again contains unreviewed results. As the programmers respond to some of the vulnerabilities identified by the audit team, the number of results begins to decrease and some of the identified vulnerabilities are fixed. At the far-right side of the graph, the growth in the number of reviewed results indicates that reviewers are beginning to look at the project again.

Summary

Building secure systems takes effort, especially for organizations that aren't used to paying much attention to security. Code review should be part of the software security process. When used as part of code review, static analysis tools can help codify best practices, catch common mistakes, and generally make the security process more efficient and consistent. But to achieve these benefits, an organization must have a well-defined code review process. At a high level, the process consists of four steps: defining goals, running tools, reviewing the code, and making fixes. One symptom of an ineffective process is a frequent descent into a debate about exploitability.

To incorporate static analysis into the existing development process, an organization needs a tool adoption plan. The plan should lay out who will run the tool, when they'll run it, and what will happen to the results. Static analysis tools are process agnostic, but the path to tool adoption is not. Take style and culture into account as you develop an adoption plan.

By tracking and measuring the security activities adopted in the development process, an organization can begin to sharpen its software security focus. The data produced by source code analysis tools can be useful for this purpose, giving insight into the kinds of problems present in the code, whether code review is taking place, and whether the results of the review are being acted upon in a timely fashion.

Index

Symbols

& (AND) operator, 412
- - (pair of hyphens), 162
/dev/random, 403
| (OR) operator, 412

A

A1 certification, 31
ABM (Analyzer Benchmark), 41
abstract interpretation, local analysis, 89
abstract syntax, building program models, 74-75
access
  back-door code, debugging, 290
  files, race conditions, 440-446
  passwords, exposing in source code, 389-391
Action class, 337
ActionForm objects, 337, 340
actions
  logging, 288
  mapping, 337
adding security reviews to existing development processes, 56-62
Address Space Layout Randomization (ASLR), 259
Adobe Reader, external entity attacks, 359-360
adoption anxiety, adding security reviews to existing development processes, 58-62
  programmers, 59
  security team, 59
AES (Advanced Encryption Stanard), 408
Ajax programming, JavaScript hijacking. *See* JavaScript hijacking
algorithms
  AES, 408
  analysis algorithms. See analysis algorithms, 83
  cryptography, 407
  implementing, 409-412
    selecting, 407-409
  passwords, encryption, 392-395
  RSA, 408
  SHA, 408
  SHA1PRNG, 399
  work-queue algorithm, 92
alias analysis, 82

_alloca() function, 271
allocation, buffer-allocation strategies, 179-180
  dynamic, 182-185
  static, 180-181
ALWAYS_TERMINATE flag, 217
analysis
  algorithms
    identifying home-grown, 412
    selecting, 409
  compilers, optimizing, 418
  cookies, 301-302
  cross-site scripting, input/output validation, 318
  debugging code, 290
  enforcing NULL after free(), 186
  entrophy, 405-407
  error messages, auditing, 326
  exception handling, 275
  exceptions, catching, 278
  Fortify Source Code Analyzer (SCA)
    applications, 478-479, 520-521
    Audit Workbench, 479-487, 521-529
    auditing, 487-491, 529-531
    C, 514-515
    customizing rules, 491-499, 531-537
    installing, 460-461, 504-505
    Java, 471-472
    results, 472-473, 475-478, 515-516, 518-520
  functions, banning, 201-203
  logging, 287
  passwords
    formatting strong, 396
    hard-coding, 390-391
    managing, 395-396
  privacy violations, 387-388
  privileged programs, 452
    open() method, 455
  PRNGs, 404
  random numbers, seeding SecureRandom, 400
  resources, managing, 285
  session identifiers, formatting, 331
  sessions, logout links, 333
  strings, libraries, 232-233
  taint propagation, 217
  unchecked return values, 268
  Validators, 345
analysis algorithms, 83
  checking assertions, 84-85
  global analysis, 91-93
  local analysis, 89
    abstract interpretation, 89
    model checking, 90
    predicate transformers, 89-90
  naïve local analysis, 85-89
  research tools, 94-95
analysis—repost under static analysis, 268
Analyzer Benchmark (ABM), 41
analyze_function(), 93
analyze_program(), 93
analyzing source code versus compiled code, 42-45
AND, 88
AND (&) operator, 412
Anderson, Ross, 389
annotations, rules, 99-100
answers to exercises
  C, 537-539
  Java, 499-501

Apache
  configuration files, validating, 122
  fix for buffer overflow, 123
API, security-enhanced API, 144
API abuse, 16
Apple, OS X (software updates that trust too much), 129
applications
  Fortify Source Code Analyzer (SCA)
    C, 520-521
    Java, 478-479
  privacy
    managing, 380-383
    violations, 383-388
  Web
    Struts Web Application Framework, 336-346
    validating input/output, 298-308, 310-318
applying
  Audit Workbench
    C, 529-531
    Java, 487-491
  POST requests, 319-321
  secrets, 416
  statistical PRNGs, 398-407
  Validators, 338-341
arc injection, 179
Arce, Ivan, 10
ARCHER, 94
architectural risk analysis, 13
architecture, JCA, 410
arguments, command-line, 426
ASLR (Address Space Layout Randomization), 259
assertions, checking with analysis algorithms, 84-85
assumptions, detecting and preventing integer overflow, 243
attack surfaces, identifying, 120
attacks
  buffer overflow, 175. *See also* buffer overflow
  cross-site request forging, 327
  cryogenic sleep, 443
  external entity attacks, Adobe Reader, 359-360
  format string, 228
  horizontal privilege escalation, 421
  privilege-escalation, 439-446, 449-452, 454-455
  session fixation, 334
  vertical privilege escalation, 421
Audit Workbench, 106
  C
    applying, 529-531
    reviewing audits, 505-512
  Java
    applying, 487-491
    Fortify Source Code Analyzer (SCA), 479-487, 521-529
  WebGoat Version 3.7, reviewing audits, 461-468
auditing
  C
    reviewing, 505-512
    source code manually, 513-514
  cookies, hidden fields, 301-302
  error messages, 326
  Java, source code manually, 469, 471
  security
    C, 529-531
    Java, 487-491

audits, 67-69
authentication, sessions, 333-336
avoiding blacklisting, input validation, 137-139

B

back-door access code, debugging, 290
backup files, debugging, 293
Baker, Jeffrey, 303
banks, phishing schemes, 316-318
banning dangerous functions, 201-203
Basic Multilingual Plane (BMP), 220
benchmarks for static analysis tools, 40-41
Berkeley fingerd daemon, 190
black-box texts, 10
blacklisting, 133
  input validation, 137-139
blacklisting, preventing cross-site scripting, 312
Blaster virus, 176
blocks, managing exceptions, 273-274
BMP (Basic Multilingual Plane), 220
BOON, 94
bound numeric input, 157-160
bounded operations, functions, 195-201, 203, 205-213
browsers. *See* interfaces, Web browsers
buffer overflow
  allocation strategies, 179-185
  Apache fix, 123
  interger overflows, 235
  Java, JNI (Java Native Interface), 254-255
  overview of, 176-178
  runtime protection, 251
    dynamic buffer overflow protections, 258-263
    dynamic protection benchmark results, 263
    safe programming languages, 251-253
    safer c dialects, 255-257
  sizes, tracking, 186-188
  strings, 189
    character sets, 218-224
    format errors, 224-228
    functions, 189-201, 203, 205-213
    libraries, 229-233
    managing null terminators, 213-218
bug finding, problem solving with static analysis, 32-33
Building Secure Software, 444

C

C
  answers to exercises, 537-539
  Audit Workbench, 521-529
    reviewing audits, 505-512
  buffer overflow, 176-178
    allocation strategies, 179-185
    tracking sizes, 186-188
  Fortify SCA
    applications, 520-521
    customizing rules, 531-537
    results, 515-520

running, 514-515
Fortify Source Code Analysis, installing, 504-505
gets(), 155
memory, unlocking values, 415
random numbers, 401-407
regular expressions, 136-137
resource leaks, preventing, 278-282
return values, checking, 266-267
runtime protection, buffer overflows, 255-257
secrets, deleting, 416
security, auditing, 529-531
source code, auditing manually, 513-514
strings, 189
character sets, 218-224
format errors, 224-228
functions, 189-201, 203, 205-213
libraries, 229-233
managing null terminators, 213-218
temporary files, security, 446-451
C++
buffer overflow, 176-178
allocation strategies, 179-185
tracking sizes, 186-188
gets(), 156
parameterized SQL, 164-167
random numbers, 401-407
regular expressions, 136-137
resource leaks, preventing, 278-282
secrets, deleting, 416
strings, 190. *See also* strings
cache, poisoning, 314
Carmargo, Luiz, 199
Cashdollar, 200
catching, exceptions, 274-275
CCured, 255
centralized logging, 286-289
time-stamp log entries, 287
CFG (context-free grammar), 73
characters, 218, 220
buffer overflow vulnerabilities, 222-224
cross-site scripting, 312
checked exceptions, 271. *See also* exceptions
maintaning, 276-278
checking returned values
in C, 266-267
in Java, 269-270
Children's Online Protection Act (COPPA), 382
chroot() function, 433-434
CL, compiler warnings (integer overflow), 244
ClassCastException, 274
classes
Action, 337
Exception, 276
File_handle, 281
java.io.File, 446
strings, 229-233
classifying vulnerabilities, 14-15
cleanse rules, 102
clear-text passwords, storing, 391-396
clear-text passwords. See also passwords, 391

code
  coupon, 399
  debugging, 286, 289-292
    backup files, 293
    Easter eggs, 293
  logging, 286
    centralized, 286-289
    time-stamp log entries, 287
  principle of least privilege, 423-432
  random numbers, 397
    C/C++, 401-407
    Java, 398-400
code points, 218
code quality, Seven Pernicious Kingdoms, 18
Code Red virus, 176
code review, performing, 48
  review cycle, 48-53
  steering clear of exploitability debates, 54-55
code values, 219
codes
  exceptions
    catching, 274-275
    maintaining checked, 276-278
    managing, 271
    top levels, 272-273
    try/finally syntax, 273-274
  return, handling errors with, 266-270
collaborative auditing, 52
Comair Airlines, integer overflow, 238-239
command injection
  preventing metacharacter vulnerabilities, 168-169
  vulnerabilties, 450
command-line arguments, 426
command-line parameters, input validation, 124-125
Common Vulnerabilities and Exposures (CVE), 176
comparing projects by severity, 65
compile-time instrumentation, runtime protection (buffer overflows), 259-261
compiled code, analyzing (versus analyzing source code), 42-45
compiler warnings, detecting and preventing integer overflow, 244-245
compilers, optimizing, 417-418
condientiality, loggers (creating to identify), 386
conditions
  privileged programs, handling unexpected, 436-438
  race, file access, 440-446
confidence, 107
confidentiality, 380. *See also* privacy
configuration
  files, storing clear-text passwords, 391-396
  Validators, 338
configuration files, input validation, 122-123
configuration information, 38
configuration rules, print, 104
connections, databases (hard-coding passwords), 389
constraint solvers, 96
context sensitivity, 83
context-free grammar (CFG), 73
context-specific defects, 14

control flow, building program models, 77-80
control flow graphs, 77
cookies
  auditing, 301-302
  session identifiers, 328-331
  sessions, 327-328
COPPA (Children's Online Protection Act), 382
copying secrets, preventing, 418-419
counterexample, 29
coupon codes, 399, 402
random numbers, 403
Coverity, 33
CQual, 94
cross-site request forging attacks, 327
cross-site scripting vulnerabilties, 303-308, 310-318
cryogenic sleep attacks, 443
CryptGenRandom() method, 401
Crypto++, 411
CryptoAPI, 402, 411
cryptographic PRNGs, 397. *See also* PRNGs
cryptography
  algorithms, 407
    implementing, 409-412
    selecting, 407, 409
  random numbers, 397
    C/C++, 401-407
    Java, 398-400
custom rules, 40
customization, rules (Fortify SCA), 491-499, 531-537
CVE (Common Vulnerabilities and Exposures), 176
Cyclone, 256

D

daemons, Berkeley fingerd, 190
data, input validation (rejecting bad data), 143-144
dataflow, building program models, 80-81
database queries, input validation, 125-127
databases, passwords (hard-coding), 389
debit cards, 384
debugging, 286, 289-292
  backup files, 293
  Easter eggs, 293
decodeFile() function, 281
decryption, passwords, 392-395
decryptPassword() method, 396
defect density, 63-64
defensive programming, 4-6
deleting secrets, 416-418
density, 63-64
  vulnerability density, measuring, 65
descriptors, standard file, 452-454
detecting integer overflow
  bad assumptions, 243
  compiler warnings, 244-245
  integer conversion rules, 245
  restricting numeric user input, 243
  sanity checks, 244
  unsigned types, 242-243
  verifying conditions for operators that can overflow, 246-249
development processes, adding security reviews to, 56, 62
  adoption anxiety, 58-62

Direct Web Remoting (DWR), 369
disabling signals, 437-438
doAlloc(), 159
document queries, XML, 362-366
doGet() method, 272
doPost() method, 272
double-free errors, 185
duplication, secrets (preventing), 418-419
DWR (Direct Web Remoting), 369
dynamic buffer allocation, 182-185
dynamic buffer overflow protections, runtime protection (buffer overflows), 258-263
dynamic protection benchmark results, runtime protection (buffer overflows), 263

E

E*Trade, 303
e-commerce
  random numbers, 402-403
  SecureRandom, 399
e-commerce Web sites, phishing schemes, 316-318
Easter eggs, 293
Eau Claire tool, 94
effective group IDs, 428
effective user IDs, 427
EJBs (Enterprise Java Beans), 385
elevating privileges, disabling signals, 437-438
eliminating unwanted results, 108-109
encapsulation, Seven Pernicious Kingdoms, 18
encoding
  characters, 218-220
    buffer overflow vulnerabilities, 222-224
    preventing cross-site scripting, 312
  UTF, 220
encryption
  algorithms, implementing, 409-412
  passwords, 392-395
enforcing
  session idle timeouts, 331-333
  trust boundaries, 131-132
enforcing NULL after free(), 186
EnterCriticalSection() function, 272
Enterprise Java Beans (EJBs), 385
entrophy, 405-407
entropy, 397
entry points, finding, 370
entry-point rules, 102
environments
  privileged programs, 426
  Seven Pernicious Kingdoms, 18
Epstein, Jeremy, 388
equivalence checking, 28
erasing passwords, 416
erasing secrets, 416-418
error handling, Seven Pernicious Kingdoms, 18
errors
  double-free, 185
  HTTP, handling, 322, 325-326
  messages, auditing, 326

privileged programs, checking, 436
recovering from, 437
with return codes, handling, 266-270
sign extension errors, integer overflow, 239-241
signed-to-unsigned conversions, integer overflow, 241-242
strings, format, 224-228
truncation, 210-213
integer overflow, 239-241
Web Services, opportunities for old errors, 370
wrap-around errors, integer overflow, 236-238
escalating privilege-escalation attacks, 439-446, 449-452, 454-455
establishing trust boundaries. *See* trust boundaries
events, handling unexpected events, 436-438
exception, IndexOutOfBoundsException, 274
Exception class, 276
exceptions
catching, 274-275
checked, maintaining, 276-278
ClassCastException, 274
InterruptedException, 277
java.lang.Exception, 271, 274
java.lang.Throwable, 325
managing, 271
top levels, 272-273
try/finally syntax, 273-274
NullPointerException, 274
RareException, 276
RuntimeException, 276
SQLException, 388
UnknownHostException, 272
execSQL() method, 386-388
executeQuery() method, 388
exercises
Audit Workbench
reviewing WebGoat Version 3.7 audits, 461-468
C
answers to, 537-539
auditing source manually, 513-514
Fortify Source Code Analyzer (SCA), 514-515
installing Fortify Source Code Analysis, 504-505
reviewing audits, 505-512
Fortify Source Code Analyzer (SCA)
applications, 478-479, 520-521
Audit Workbench, 479-487, 521-529
auditing, 487-491, 529-531
customizing rules, 491-499, 531-537
results, 472-473, 475-478, 515-520
Java
answers to, 499-501
auditing source manually, 469-471
Fortify Source Code Analyzer (SCA), 471-472
installing Fortify Source Code Analysis, 460-461
Exploiting Software, 176

exploits. *See also* attacks
  arc injection, 179
  buffer overflow, 176-178
    allocation strategies, 179-185
    tracking sizes, 186-188
  format string, 228
  return-into-libc, 179
exposing passwords in source code, 389-391
exploitability debates, steering clear of, 54-55
ext, 88
external references, XML, 358-362

F

fake Web sites, 316. *See also* phishing
false alarms, 23
false negatives, 23
false paths, 88
false positives, 23, 105
Federal Information Security Management Act (FISMA), 382
fgets() function, 266
file systems, privileged programs, 427
filenames, unique, 447-448
files
  access, race conditions, 440-446
  backup, debugging, 293
  configuration, storing clear-text passwords, 391-396
  standard file descriptors, 452-454
  temporary, security, 446-451
  unique, 448-449
filesystems
  functions, TOCTOU vulnerabilities, 442
  privileges, restricting on, 433-435
File_handle class, 281
finally blocks, managing exceptions, 273-274
FindBugs, 33, 100
finding entry points, Web Services, 370
FISMA (Federal Information Security Management Act), 382
fixation, limiting session, 334
flag functions, security-enhanced API, 152
flags
  ALWAYS_TERMINATE, 217
  hard-coded passwords, 391
  privayc violations, 386
flow-insensitive analysis, 89
form bean mapping, 337
formal verification, 31
formatting
  session identifiers, 329-331
  strings, errors, 224-228
  unique filenames, 447-448
forms, validator, 337
Fortify rule, 98
Fortify Software, 34
Fortify Source Code Analyzer (SCA)
  C
    applications, 520-521
    Audit Workbench, 521-529
    auditing, 529-531
    customizing rules, 531-537
    installing, 504-505
    results, 515-520
    running, 514-515

Java
applications, 478-479
Audit Workbench, 479-487
auditing, 487-491
customizing rules, 491-499
installing, 460-461
results, 472-478
running, 471-472
Foundations of AJAX, 304
free() function, 185
enforcing NULL after, 186
Fujaba, 27
Fujitsu, 384
function summaries, 92
functions
banning dangerous, 201-203
chroot(), 433-434
decodeFile(), 281
EnterCriticalSection(), 272
fgets(), 266
filesystems, TOCTOU vulnerabilities, 442
free(), 185
enforcing NULL after, 186
GetTempFileName(), 447
Helper, sizing buffers, 188
malloc(), 185
mkstemp(), 449
read(), 269
strings
bounded operations, 195-201, 203, 205-213
gets(), 189
reimplementation, 194
scanf(), 190-191
sprintf(), 193
strcpy(), 192
strlcat(), 196
strlcpy(), 196
strncat(), 200, 208
strncpy(), 204
temporary files, security, 446, 448-449
Thread.sleep(), 277
tmpfile(), 449
_alloca(), 271
fuzzing, 11

G

Gaim instant-messaging client, 205
GCC, compiler warnings (integer overflow), 245
generateSeed() method, 400
generating random numbers
C/C++, 401-407
Java, 398-400
generic defects, 14
Get requests, 319-321
getConnection() method, 390-391
gets() function, 155, 189
GetTempFileName() function, 447
GLBA (Gramm-Leach-Bliley Act), 382
global analysis, 83, 91-93
goals, establishing for code review, 49-50
Gramm-Leach-Bliley Act (GLBA), 382
grouping results, 106-108

H

halting problem, 35
handling
errors
HTTP, 322, 325-326
with return code, 266-270
unexpected events, 436-438

hard-coding passwords, 389
hardened system libraries, runtime protection (buffer overflows), 262
headers
  HTTP, 301-302
  referer, 327
  User-Agent, input validation, 299-302
Health Insurance Portability and Accountability Act (HIPAA), 382
helper functions, sizing buffers, 188
hidden fields, Show Source option, 302
hijacking Web pages, 314
HIPAA (Health Insurance Portability and Accountability Act), 382
holding secrets, minimizing time, 414-415
horizontal privilege escalation attacks, 421
Howard, Michael, 13, 52
HTTP
  headers, 301-302
  requests, validating, 298-299, 301-308, 310-318
  responses, splitting, 314-315
  security, 319
    applying POST, 319-321
    handling errors, 322, 325-326
    maintaining session state, 328-336
    ordering requests, 322-323
    request provenance, 327-328
  traffic sniffers, 302
Huseby, Sverre H., 359-360

I

identifiers
  random request, 328
  session, 328-331
identifying attack surfaces, 120
identities, phishing, 316-318
IDEs (integrated development environments), 27
idle timeouts, enforcing session, 331-333
image-display software, vulnerabilities, 8
impersonating servers, 129
implementation, algorithms, 409-412
inbound passwords, 388
IndexOutOfBoundsException, 274
indirect selection, 133
  input validation, 133-134
injection, command vulnerabilities, 450-452
input
  Struts Web Application Framework, 336
    analyzing Validators, 345
    applying Validators, 338-341
    configuring Validators, 338
    maintaining validation logic, 343-346
    validating parameters, 342
  transforming, 140-141
  validating, 298-299, 301-308, 310-318
input length, checking, 153-156
input validation, 119
  blacklisting, 137-139
  bound numeric input, 157-160

check input length, 153-156
establishing trust boundaries, 130-131
good input validation as the default, 144-152
mistaking usability for security, 142
rejecting bad data, 143-144
strong input validation, 133-134
validating all input, 120
validating input from all sources, 121-122
command-line parameters, 124-125
configuration files, 122-123
database queries, 125-127
network services, 127-128
Web Services, 366-368
whitelisting, 135-136
input validation and representation, 16
installation, Fortify Source Code Analysis
C, 504-505
Java, 460-461
integer conversion rules, detecting and preventing integer overflow, 245
integer overflow, 236-237
Comair Airlines, 238-239
conversion between signed and unsigned data types, 241-242
detecting and preventing
bad assumptions, 243
compiler warnings, 244-245
integer conversion rules, 245
restricting numberic user input, 243
sanity checks, 244
unsigned types, 242-243
verifying conditions for operators that can overflow, 246-249
Java, 157-158
sign extension errors, 239-241
truncation errors, 239-241
user input, 250
wrap-around errors, 236-238
integer overflows, 235
integers, 236
integral user input, interger overflow vulnerabilities, 250
integrated development environments (IDEs), 27
interfaces, SingleThreadModel, 385
International Organization for Standardization (ISO), 220
Internet Explorer High Encryption Pack, 411
interprocedural analysis, 83
InterruptedException, 277
intraprocedural analysis, 83
IntSafe, 246-248
ISO (International Organization for Standardization), 220

J

Java
answers to exercises, 499-501
Audit Workbench, 479, 481, 483-487
bounds checking, 156

Java, *continued*
  buffer overflows, JNI (Java Native Interface), 254-255
  cryptography, 393
  Fortify SCA
    applications, 478-479
    customizing rules, 491-499
    results, 472-478
    running, 471-472
  Fortify Source Code Analysis, installing, 460-461
  gets(), 156
  hard-coding passwords, 389
  integer overflow, 157-158
  passwords, storing clear-text passwords, 392
  random numbers, 398-400
  resource leaks, preventing, 283-285
  return values, checking, 269-270
  security, auditing, 487-491
  source code, auditing manually, 469-471
  Web applications
    HTTP, 319-328
    input/output validation, 298-299, 301-308, 310-318
    maintaining session state, 328-336
Java Cryptography Architecture (JCA), 410
Java Cryptography Extension (JCE), 409-410
Java Modeling Language (JML), 99
Java Native Interface (JNI), 254-255
java.io.File class, 446
java.lang.Exception, 271, 274
java.lang.Throwable, 325
java.util.logging package, 286
java.util.Random, 274
JavaScript hijacking, Web Services, 370-375
  preventing direct execution of responses, 375-376
JavaScript Object Notation (JSON), 371
JavaServer Pages (JSP), 43
JCA (Java Cryptography Architecture), 410
JCE (Java Cryptography Extension), 409-410
JML (Java Modeling Language), 99
JNI (Java Native Interface), 254-255
JSON (JavaScript Object Notation), 371

K

keys, secret, 393-395
Klockwork, 33

L

LAPSE (Lightweight Analysis for Program Security in Eclipse), 94
leaks, resource (preventing), 278-285
least privilege, principle of, 423-432
lexical analysis, 72-73
libraries
  hardened system libraries, runtime protection (buffer overflows), 262
  strings, 229-233
Libsafe, 262

Libverify, 262
lifetimes, maximum session, 331-333
limitations of static analysis, 23
limiting session fixation, 334
links, logout, 333
Linux, privileges, 423, 427-432
Lipner, Steve, 13, 48
Livshits, Benjamin, 40
local analysis, 83, 89
  abstract interpretation, 89
  model checking, 90
  predicate transformers, 89-90
log forging, preventing meta-character vulnerabilities, 169-172
loggers, 388
  creating, 386
logging, 286
  centralized, 286-289
    time-stamp log entries, 287
  queries, 384-385
logic, validation, 343-346
login, privileges, 428
logout links, 333
lowering, 75

M

malloc(), 159, 185
management
  exceptions, 271
    catching, 274-275
    maintaining checked, 276-278
    top levels, 272-273
    try/finally syntax, 273-274
  null terminators, 213-218
  passwords, 395-396
    formatting strong, 396
  privacy, 380-383
    violations, 383-385, 387-388
  privileged programs, 427
    applying principle of least privilege, 427-432
    handling unexpected events, 436-438
    privilege-escalation attacks, 439-446, 449-452, 454-455
    restricting on filesysytems, 433-435
  requests, ordering, 322-323
  resources, preventing leaks, 278-285
  session state, 328-336
  validation logic, 343-346
manual bounded operation checks, 199
manual null-terminate, bounded operations, 207
maps
  ActionForm objects, 340
  actions, 337
  form bean mapping, 337
maximum session lifetimes, 331-333
McGraw, Gary, 13
measuring vulernability density, 65
memory
  buffer overflow, 176-178
    allocation strategies, 179-185
    tracking sizes, 186-188
  nonexecutable memory segments, 258-259
  secrets, 412-413
    applying, 416
    deleting, 416-418
    minimizing time holding, 414-415
    preventing duplication, 418-419

memory safety, 251
memset() method, 417-418
messages, auditing errors, 326
metacharacters, preventing vulnerabilities, 160-161
  command injection, 168-169
  log forging, 169-172
  parametereized requests, 161-166
  path manipulation, 167-168
methods
  CryptGenRandom(), 401
  decryptPassword(), 396
  execSQL(), 386, 388
  executeQuery(), 388
  generateSeed(), 400
  getConnection(), 391
  memset(), 417-418
  open(), 455
  Random.nextInt(), 398
  rand_s(), 402
  realloc(), 419
  RtlGenRandom(), 402
  Utils.processHost(), 272
methods,doGet(), 272
methods,doPost(), 272
metrics
  process metrics, 67-69
  static analysis metrics, 62
    breaking down results by category, 66
    comparing projects by severity, 65
    measuring vulnerability density, 65
    monitoring trends, 66
    process metrics, 67-69
Microsoft Passport, debugging, 291-292
Microsoft CryptoAPI, 411
mkstemp() function, 449
model checking, local analysis, 90
model-checking rules, print, 104
Model-View-Controller (MVC) pattern, 336
modifying memory, buffer overflow, 176-178
MOPS (Model Checking Programs for Security properties), 95
Morris Worm, 176, 190
MVC (Model-View-Controller) pattern, 336
MySpace, cross-site scripting, 309-312

N

naïve local analysis, analysis algorithms, 85-89
names, unique filenames, 447-448
Nettle, 411
network services, input validation, 127-128
Nimda virus, 176
nonexecutable memory segments, runtime protection (buffer overflows), 258-259
NOT, 88
notation, 88
NULL, free() function (enforcing after), 186
null terminators, managing, 213-218
NullPointerException, 274
numbers, random, 274, 397
  C/C++, 401-407
  Java, 398-400

numeric user input, detecting and preventing integer overflow, 243

O

objects, ActionForm, 337, 340
open redirects, 316-318
Open Web Application Security Project (OWASP), 19, 461
open() method, 455
operators
  AND (&), 412
  OR (|), 412
  sizeof, 186
optimization, compilers, 417-418
options, Show Source, 302
OR, 88
OR (|) operator, 412
Orange Book, 31
ordering requests, 322-323
Ounce Labs, 34
outbound passwords, 388
  clear-text passwords, storing, 391-396
  source code, exposing in, 389-391
output, validating, 298-299, 301-308, 310-318
over exposure, Web Services, 369
OWASP (Open Web Application Security Project), 19, 461
OWASP Guide to Building Secure Web Applications, 312

P

packages, java.util.logging, 286
parameterized requests, preventing metacharacter vulnerabilities, 161-166
parameters, validating, 342
parse trees, 73
parsers, standards-compliant XML parsers, 350-352
parsing, building program models, 73-74
partial specification, 29
pass-through rules, 102
Passport, debugging, 291-292
passwords
  decryption, 392-395
  encryption, 392-395
  erasing, 416
  formatting, 396
  inbound, 388
  managing, 395-396
  outbound, 388
    exposing in source code, 389-391
    storing clear-text passwords, 391-396
patches, updating passwords, 389
path manipulation, preventing metacharacter vulnerabilities, 167-168
patterns, MVC, 336
Payment Card Industry (PCI) Data Security Standard, 382-384, 392
PCI (Payment Card Industry) Data Security Standard, 382-384, 392
penetration tests, 10
performing code review, 48
  review cycle, 48-53

steering clear of exploitability debates, 54-55
personal identification numbers (PINs), 384
Petroski, Henry, 3
phishing, 304, 316-318
Phishing and Countermeasures, 316
PID (process ID), 403
PINs (personal identification numbers), 384
Pixy, 95
pointer aliasing, building program models, 82-83
pointers, volatile, 417
POST requests, applying, 319-321
postcondition, 105
PQL (Program Query Language), 101
Practical Cryptography, 407
precondition, 104
predicate transformers, local analysis, 89-90
preventing
  direct execution of responses, JavaScript hijacking, 375-376
  integer overflow
    bad assumptions, 243
    compiler warnings, 244-245
    integer conversion rules, 245
    restricting numeric user input, 243
    sanity checks, 244
    unsigned types, 242-243
    verifying conditions for operators that can overflow, 246-249
  metacharacter vulnerabilities, 160-161
    command injection, 168-169
    log forging, 169-172
    parameterized requests, 161-166
    path manipulation, 167-168
  secret duplication, 418-419
prevention
  buffer overflow
    allocation strategies, 179-185
    overview of, 176-178
    tracking sizes, 186-188
  cross-site scripting, 312-314
  resource leaks, 278-285
principle of least privilege, 423-432
print, rules, 103
  configuration rules, 104
  model-checking rules, 104
  structural rules, 104
  taint-propagation rules, 104-105
privacy, 380
  managing, 380-383
  violations, 383-388
Privilege Separated OpenSSH project, 433
privilege-escalation attacks, 439-440, 442-444, 446, 449-452, 454-455
privileged programs, 421-423
  command-line arguments, 426
  environments, 426
  file systems, 427
  managing, 427
    applying principle of least privilege, 427-432

handling unexpected events, 436-438
restricting on filesystems, 433-435
principle of least privilege, 423-432
privilege-escalation attacks, 439-440, 442-444, 446, 449-452, 454-455
PRNGs (pseudo-random number generators), 397
C/C++, generating, 401-407
Java, generating, 398-400
process ID (PID), 403
process metrics, 67-69
profiles, privileges, 424
program models, building, 72
abstract syntax, 74-75
lexical analysis, 72-73
parsing, 73-74
pointer aliasing, 82-83
semantic analysis, 76
taint propagation, 82
tracking control flow, 77-80
tracking data flow, 80-81
Program Query Language (PQL), 101
program slicing, 88
program understanding, problem solving with static analysis, 27
program verification, problem solving with static analysis, 28-31
programmers, adoption anxiety (adding security reviews to existing development processes), 59
Programming Jakarta Struts, 2nd Edition (italics), 336
programming languages, runtime protection (buffer overflows), 251-253
programs
privacy
managing, 380-383
violations, 383-388
privleged, 421-423
applying principle of least privilege, 427-432
command-line arguments, 426
environments, 426
file systems, 427
handling unexpected events, 436-438
managing, 427
principle of least privilege, 423-432
privilege-escalation attacks, 439-440, 442-444, 446, 449-452, 454-455
restricting on filesystems, 433-435
projects, comparing by severity, 65
propagation, taint, 387-388
property checking, problem solving with static analysis, 28-31
ProPolice, 259
protecting private data, 382. *See also* privacy
protection, Web browsers, 303-308, 310-318

provenance, requests, 327-328
pseudo-random number generators. *See* PRNGs
publicy vetted algorithms, implementing, 409-412

Q

quality, 9
  testing, 9-11
queries
  database queries, input validation, 125-127
  document queries, XML, 362-366
  logging, 384-385
  privacy violations, 388

R

race conditions, file access, 440-446
RAII (Resource Acquisition Is Initialization), 281
random numbers, 274, 397
  C/C++, 401-407
  Java, 398-400
random request identifiers, 328
Random.nextInt() method, 398
rand_s() method, 402
RareException, 276
read() function, 269
readlink(), 146-149
real group IDs, 428
real user IDs, 427
realloc() method, 419
reasons for not fixing bad code, 55
recovering from errors, 437
recovering passwords, 389
references, external references (XML), 358-362
referer headers, 327
reflected cross-site scripting. 306. *See also* cross-site scripting
regular expressions in C and C++, 136-137
regulation, 380
reimplementation, functions, 194
rejecting bad data, input validation, 143-144
Remote Procedure Call (RPC), 349
reporting results, 105-106
  eliminating unwanted results, 108-109
  grouping and sorting, 106-108
  significance of results, 109-113
requests
  GET, 319-321
  HTTP, validating, 298-299, 301-308, 310-318
  ordering, 322-323
  POST, applying, 319-321
  provenance, 327-328
  random identifiers, 328
requirements
  privacy, 380
    managing, 380-383
    violations, 383-388
  privileges, 424
research tools, analysis algorithms, 94-95
resizing buffers, 187
Resource Acquisition Is Initialization (RAII), 281

resources, preventing leaks, 278-285
responses, HTTP (splitting), 314-315
restrictions, privileges on filesystems, 433-435
results
  breaking down by categories, 66
  Fortify Source Code Analyzer (SCA)
    Audit Workbench, 479-487, 521-529
    C, 515-520
    Java, 472-478
  reporting, 105-106
    elmininating unwanted results, 108-109
    grouping and sorting, 106-108
    significance of results, 109-113
return codes, handling errors, 266-270
return-into-libc, 179
review cycle, performing code review, 48-53
  establishing goals, 49-50
  making fixes, 53
  running static analysis tools, 50-51
reviewing audits, C, 505-512
Rice's theorem, 35
Rice, Henry, 35
Rivest, Shamir, and Adleman. *See* RSA algorithm
RPC (Remote Procedure Call), 349
RSA (Rivest, Shamir, and Adleman) algorithm, 408
RtlGenRandom() method, 401-402
rule formats, 97, 100-101
  annotations, 99-100
  specialized rule files, 97-98
rule sets, 40
rules, 96-97
  Fortify, 98
  Fortify SCA, customizing, 491-497, 499, 531-537
  Functions, banning, 201-203
  print, 103
    configuration rules, 104
    model-checking rules, 104
    structural rules, 104
    taint-propagation rules, 104-105
  rule formats, 97, 100-101
    annotations, 99-100
    specialized rule files, 97-98
  taint propagation, 101-103
running Fortify SCA
  C, 514-515
  Java, 471-472
runtime protection, buffer overflows, 251
  dynamic buffer overflow protections, 258-263
  dynamic protection benchmark results, 263
  safe programming languages, 251-253
  safer C dialects, 255-257
RuntimeException, 276

S

Safe Harbor Privacy Framework, 382

SafeInt, 246
safety, bounded operations, 205-206
SAL (Standard Annotation Language), 100
SAMATE group, 41
sanity checks, detecting and preventing integer overflow, 244
Sasser virus, 176
SATURN, 95
saved group IDs, 428
saved user IDs, 427
SCA (Fortify Source Code Analyzer), 98
  C
    applications, 520-521
    Audit Workbench, 521-529
    auditing, 529-531
    customizing rules, 531-537
    results, 515-520
    running, 514-515
  Java
    applications, 478-479
    Audit Workbench, 479-487
    auditing, 487-491
    customizing rules, 491-497, 499
    results, 472-478
    running, 471-472
scanf() function, 190-191
scripting cross-site, 303-308, 310-318
secret keys, 393-395
secrets
  decryption, 392-395
  encryption, 392-395
  memory, 412-413
    applying, 416
    deleting, 416-418
    minimizing time holding, 414-415
    preventing duplication, 418-419
  random numbers, 397
    C/C++, 401-407
    Java, 398-400
Secure Hash Algorithm (SHA), 408
SecureRandom, 399
security
  auditing
    C, 529-531
    Java, 487-491
  HTTP, 319
    applying POST requests, 319-321
    handling errors, 322-326
    maintaining session state, 328-336
    ordering requests, 322-323
    request provenance, 327-328
  input validation, 142
  passwords
    exposing in source code, 389-391
    outbound, 388
    storing clear-text passwords, 391-396
  random numbers, 397
    C/C++, 401-407
    Java, 398-400
  temporary files, 446-451
Security Engineering, 389
security features, 6-9
  Seven Pernicious Kingdoms, 17

security reviews
  adding to existing development processes, 56, 62
    adoption anxiety, 58-62
  examples of, 56-57
  problem solving with static analysis, 33-35
security teams, adoption anxiety (adding security reviews to existing development processes), 59
security-enhanced API, 144
  flag functions, 152
selection, algorithms, 407-409
semantic analysis, building program models, 76
semantic checks, 120
Sendmail 8.10.1, privileges, 436
servers, impersonating, 129
Service-Oriented Architecture (SOA), 349
services, network services (input validation), 127-128
Servlet
  privacy violations, 385-386
  SingleThreadModel interface, 385
session fixation attacks, 334
sessions
  authentication, 333-336
  cookies, 327-328
  maximum lifetimes, 331-333
  random numbers, 397
    C/C++, 401-407
    Java, 398-400
  state, maintaining, 328-336
setuid root, 424-426
Seven Pernicious Kingdoms, 15-16
  API abuse, 16
  code quality, 18
  encapsulation, 18
  environment, 18
  error handling, 18
  input validation and representation, 16
  security features, 17
  time and state, 17
  vulnerabilities, 19
severity, 107
  rules, 105
SHA (Secure Hash Algorithm), 408
SHA1PRNG algorithm, 399
The Shellcoder's Handbook, 176
Show Source option, 302
Siegel, Aaron, 77
sign extension errors, integer overflow, 239-241
signals, disabling, 437-438
signed data types, integer overflow, 241-242
Simplify, 96
SingleThreadModel interface, 385
sink rules, 101
sizeof operator, 186
sizes, buffer (tracking), 186-188
Slammer virus, 176
SMTP daemon qwik-smtpd, reviewing audit of, 505-512
sniffers, traffic, 302
SOA (Service-Oriented Architecture), 349
software security, 4
software-development methodologies, steps of, 11-13
solving problems with static analysis, 24

bug finding, 32-33
program understanding, 27
program verification, 28-31
property checking, 28-31
security review, 33-35
style checking, 26-27
type checking, 24-25
sorting results, 106-108
source code, 460. *See also* code
analyzing, versus compiled code, 42-45
C, auditing manually, 513-514
Fortify Source Code Analyzer (SCA)
applications, 478-479, 520-521
Audit Workbench, 479-487, 521-529
auditing, 487-491, 529-531
C, 514-515
customizing rules, 491-499, 531-537
installing, 460-461, 504-505
Java, 471-472
results, 472-478, 515-520
Java, auditing manually, 469-471
passwords, exposing in, 389-391
Source Code Analyzer (SCA)
C
applications, 520-521
Audit Workbench, 521-529
auditing, 529-531
customizing rules, 531-537
results, 515-520
running, 514-515
Java
applications, 478-479
Audit Workbench, 479-487
auditing, 487-491
customizing rules, 491-497, 499
results, 472-478
running, 471-472
source rules, 101
specialized rule files, 97-98
Splint, 95
splitting, HTTP response, 314-315
sprintf() function, 193
SQL, parameterized SQL in C++, 164-167
SQLException, 388
SSP (Stack Smashing Protection), 259
StackGuard, 260
stacks, buffer overflow, 178
StackShield, 261
Standard Annotation Language (SAL), 100
standard file descriptors, 452-454
starting sessions upon authentication, 333-336
state, maintaining sessions, 328-336
static analysis, 3
benchmarks, 40-41
capabilities of, 22-23
limitations of, 23
methodologies, 11-13
solving problems with, 24
bug finding, 32-33
program understanding, 27
program verification, 28-31
property checking, 28-31
security review, 33-35
style checking, 26-27

type checking, 24-25
success criteria, 36-37
ease of use, 41-42
finding the right stuff, 40
programs, understanding, 37-38
trade-offs, 38-39
static analysis metrics, 63
breaking down results by categories, 66
comparing projects by severity, 65
measuring vulnerability density, 65
monitoring trends, 66
process metrics, 67-69
static buffer allocation, 180-181
statistical PRNGs, applying, 398-407. *See also* PRNGs
std::string class, 229
storage, clear-text passwords, 391-396
stored cross-site scripting, 308. *See also* cross-site scripting
strategies, buffer-allocation, 179-180
dynamic, 182-185
static, 180-181
strcpy() function, 192
strcpy(), 34
string passwords, formatting, 396
strings, 189
buffer overflow, 175
functions
bounded operations, 195-203, 205-213
character sets, 218-224
format errors, 224-228
gets(), 189
managing null terminators, 213-218
reimplementation, 194
scanf(), 190-191
sprintf(), 193
strcpy(), 192
libraries, 229-233
strlcat() function, 196
strlcpy() function, 196
strncat() function, 200, 208
strncpy() function, 204
structural analysis, 76
structural rules
checking, 77
print, 104
Struts in Action, 336
Struts Web Application Framework, 336
logic, maintainig validation, 343-346
parameters, validating, 342
Validator
applying, 338-341
configuring, 338
static analysis, 345
style checking, problem solving with static analysis, 26-27
success criteria, 36-37
ease of use, 41-42
finding the right stuff, 40
programs, understanding, 37-38
trade-offs, 38-39
surrogate pairs, 219
symbolic simulation, 87
syntax checks, 120
system identifiers, 358

T

taint flags, 103
taint propagation, 217, 387-388
  building program models, 82
  rules, 101-105
TCSEC (Trusted Computer System Evaluation Criteria), 31
temporal safety properties, 29
temporary files, security, 446-451
termination, null (managing), 213-218
test, penetration tests, 10
testing, 9-11
  fuzzing, 11
tests, black-box texts, 10
The Unicode Standard, 218
Thread.sleep() function, 277
throwing exceptions, checked, 276-278
time and state, Seven Pernicious Kingdoms, 17
time-of-check, time-of-use (TOCTOU) race conditions, 440-446
time-stamp log entires, 287
timeouts, sessions (enforcing idle), 331-333
tmpfile() function, 449
TOCTOU (time-of-check, time-of-use) race conditions, 440-446
Tomcat Servlet Container, 274
top levels, managing exceptions, 272-273
tracking
  buffer sizes, 186-188
  control flow, building program models, 77-80
  data flow, building program models, 80-81
  privacy violations, 387-388
traffic, sniffers, 302
transforming input, 140-141
transitions, privilege profiles, 424
trends, monitoring, 66
truncation
  errors, 210-213
    integer overflow, 239-241
  static buffer allocation strategies, 181
trust
  Apple OS X, software updates that trust too much, 129
  privileged programs, 426-427
trust boundaries
  enforcing, 131-132
  input validation, 130-131
TRUSTe, 387
Trusted Computer System Evaluation Criteria (TCSEC), 31
try blocks, managing exceptions, 273-274
Turing, Alan, 35
type checking, problem solving with static analysis, 24-25
type safety, 251
types, buffer overflow, 176-178
  allocation strategies, 179-185
  tracking sizes, 186-188

U

UAC (User Account Control), 423
UCS (Universal Character Set), 220
unchecked exceptions, 271. *See also* exceptions

undecidability, 35
unexpected events, handling, 436-438
Unicode Transformation Format (UTF), 220
unique filenames, 447-448
unique files, 448-449
Universal Character Set (UCS), 220
UNIX, privileges, 423, 427-432
UnknownHostException, 272
unlocking values in memory, 415
unsigned data types
  detecting and preventing integer overflow, 242-243
  integer overflow, 241-242
unterminated strings, 217
updating passwords, 389
usability, input validation, 142
User Account Control (UAC), 423
user input, integer overflow, 250
User-Agent headers, input validation, 299-302
UTF (Unicode Transformation Format), 220
Utils.processHost() method, 272

V

validation
  bounded string operations, 197-198
  input validation
    blacklisting, 137-139
    bound numeric input, 157-160
    check input length, 153-156
    establishing trust boundaries, 130-131
    good input validation as the default, 144-152
    mistaking usability for security, 142
    rejecting bad data, 143-144
    strong input validation, 133-134
  logic, 343-346
  parameters, 342
  XML, 352-357
validator forms, 337
Validators
  applying, 338-341
  configuring, 338
  static analysis, 345
values
  code, 219
  memory, unlocking, 415
  return
    checking in C, 266-267
    checking in Java, 269-270
verifying conditions for operators that can overflow, 246-249
vertical privilege escalation attacks, 421
viewing
  filesystems, 433
  hard-coded passwords, 390-391
  Show Source option, 302
violations, privacy, 383-388
virtual execution environments, runtime protection (buffer overflows), 262
viruses, buffer overflow, 176. *See also* buffer overflow
volatile pointers, 417
Von Neumann, John, 397
Vstr library, 230

vulnerabilities
  buffer overflow. *See* buffer overflow
  classifying, 14-15
  context-specific defects, 14
  cross-site scripting, 303-308, 310-318
  Easter eggs, 293
  generic defects, 14
  in image-display software, 8
  preventing metacharacter vulnerabilities, 160-161
    command injection, 168-169
    log forging, 169-172
    parameterized requests, 161-166
    path manipulation, 167-168
  Seven Pernicious Kingdoms
    API abuse, 16
    code quality, 18
    encapsulation, 18
    environment, 18
    error handling, 18
    input validation and representation, 16
    security features, 17
    time and state, 17
    vulnerabilities, 19
  strings, 189
    character sets, 218-224
    format errors, 224-228
    functions, 189-201, 203, 205-213
    libraries, 229-233
    managing null terminators, 213-218
  temporary files, 446, 448-449
vulnerability density, measuring, 65
vulnerability dwell, 67
vulnerabilities. *See also* attacks
  command injection, 450-452
  TOCTOU, 440-446

W

Wagner, David, 176, 397
Wall Street Journal, 384
weak session identifiers, 331
weakest precondition (WP), 89
Web applications
  Java
    HTTP, 319-328
    input/output validation, 298-299, 301-308, 310-318
    maintaining session state, 328-336
  Struts Web Application Framework, 336
    analyzing Validators, 345
    applying Validators, 338-341
    configuring Validators, 338
    maintaining validation logic, 343-346
    validating parameters, 342
Web browsers
  input validation, 299-302
  protecting, 303-308, 310-318
  Sow Source option, 302
Web pages, hijacking, 314
Web Services, 349, 366
  DWR, 369
  entry points, finding, 370
  input validation, 366-368
  JavaScript hijacking, 371-375

preventing direct execution of responses, 375-376
opportunities for old errors, 370
over exposure, 369
WSDL, 368-369
Web Services Description Language (WSDL), 368-369
Web sites, phishing, 316-318
WebGoat Version 3.7, reviewing audits, 461-468
WebMethods, 388
WebSphere Application Server (version 6.1), 392
whitelisting, input validation, 135-136
whole-program analysis, 92
Wilander, John, 263
Windows Vista, UAC, 423
work-queue algorithm, 92
worms
cross-site scripting, 309-312
Morris Worm, 176, 190
WP (weakest precondition), 89
wrap-around errors, integer overflow, 236-238
WSDL (Web Services Description Language), 368-369

X

xg++, 95
XML (Extensible Markup Language), 349-350
document queries, 362-366
external references, 358-362
standards-compliant XML parsers, 350-352
validation, 352-357
XML injection, 354
XML Schema, 355
XPath injection, 362
XySSL, 411

Z

Zotob virus, 176