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.