SEI SERIES • A CERT® BOOK

Secure Coding in C and C++

Second Edition

Robert C. Seacord

Foreword by Richard D. Pethia
CERT Director

Secure Coding in C and C++

Second Edition

The SEI Series in Software Engineering

Software Engineering Institute of Carnegie Mellon University and Addison-Wesley

Visit **informit.com/sei** for a complete list of available publications.

The SEI Series in Software Engineering is a collaborative undertaking of the Carnegie Mellon Software Engineering Institute (SEI) and Addison-Wesley to develop and publish books on software engineering and related topics. The common goal of the SEI and Addison-Wesley is to provide the most current information on these topics in a form that is easily usable by practitioners and students.

Titles in the series describe frameworks, tools, methods, and technologies designed to help organizations, teams, and individuals improve their technical or management capabilities. Some books describe processes and practices for developing higher-quality software, acquiring programs for complex systems, or delivering services more effectively. Other books focus on software and system architecture and product-line development. Still others, from the SEI's CERT Program, describe technologies and practices needed to manage software and network security risk. These and all titles in the series address critical problems in software engineering for which practical solutions are available.

Make sure to connect with us!
informit.com/socialconnect

Secure Coding in C and C++

Second Edition

Robert C. Seacord

Addison-Wesley

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

Software Engineering Institute | Carnegie Mellon

The SEI Series in Software Engineering

Visit us on the Web: informit.com/aw

Library of Congress Cataloging Control Number: 2013932290

ISBN-13: 978-0-321-82213-0
ISBN-10: 0-321-82213-7
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
First printing, March 2013

To my wife, Rhonda, and our children, Chelsea and Jordan

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

Society's increased dependency on networked software systems has been matched by an increase in the number of attacks aimed at these systems. These attacks—directed at governments, corporations, educational institutions, and individuals—have resulted in loss and compromise of sensitive data, system damage, lost productivity, and financial loss.

While many of the attacks on the Internet today are merely a nuisance, there is growing evidence that criminals, terrorists, and other malicious actors view vulnerabilities in software systems as a tool to reach their goals. Today, software vulnerabilities are being discovered at the rate of over 4,000 per year. These vulnerabilities are caused by software designs and implementations that do not adequately protect systems and by development practices that do not focus sufficiently on eliminating implementation defects that result in security flaws.

While vulnerabilities have increased, there has been a steady advance in the sophistication and effectiveness of attacks. Intruders quickly develop exploit scripts for vulnerabilities discovered in products. They then use these scripts to compromise computers, as well as share these scripts so that other attackers can use them. These scripts are combined with programs that automatically scan the network for vulnerable systems, attack them, compromise them, and use them to spread the attack even further.

With the large number of vulnerabilities being discovered each year, administrators are increasingly overwhelmed with patching existing systems. Patches can be difficult to apply and might have unexpected side effects. After

a vendor releases a security patch it can take months, or even years, before 90 to 95 percent of the vulnerable computers are fixed.

Internet users have relied heavily on the ability of the Internet community as a whole to react quickly enough to security attacks to ensure that damage is minimized and attacks are quickly defeated. Today, however, it is clear that we are reaching the limits of effectiveness of our reactive solutions. While individual response organizations are all working hard to streamline and automate their procedures, the number of vulnerabilities in commercial software products is now at a level where it is virtually impossible for any but the best-resourced organizations to keep up with the vulnerability fixes.

There is little evidence of improvement in the security of most products; many software developers do not understand the lessons learned about the causes of vulnerabilities or apply adequate mitigation strategies. This is evidenced by the fact that the CERT/CC continues to see the same types of vulnerabilities in newer versions of products that we saw in earlier versions.

These factors, taken together, indicate that we can expect many attacks to cause significant economic losses and service disruptions within even the best response times that we can realistically hope to achieve.

Aggressive, coordinated response continues to be necessary, but we must also build more secure systems that are not as easily compromised.

About Secure Coding in C and C++

Secure Coding in C and C++ addresses fundamental programming errors in C and C++ that have led to the most common, dangerous, and disruptive software vulnerabilities recorded since CERT was founded in 1988. This book does an excellent job of providing both an in-depth engineering analysis of programming errors that have led to these vulnerabilities and mitigation strategies that can be effectively and pragmatically applied to reduce or eliminate the risk of exploitation.

I have worked with Robert since he first joined the SEI in April, 1987. Robert is a skilled and knowledgeable software engineer who has proven himself adept at detailed software vulnerability analysis and in communicating his observations and discoveries. As a result, this book provides a meticulous treatment of the most common problems faced by software developers and provides practical solutions. Robert's extensive background in software development has also made him sensitive to trade-offs in performance, usability, and other quality attributes that must be balanced when developing secure

code. In addition to Robert's abilities, this book also represents the knowledge collected and distilled by CERT operations and the exceptional work of the CERT/CC vulnerability analysis team, the CERT operations staff, and the editorial and support staff of the Software Engineering Institute.

—Richard D. Pethia
CERT Director

This page intentionally left blank

Preface

CERT was formed by the Defense Advanced Research Projects Agency (DARPA) in November 1988 in response to the Morris worm incident, which brought 10 percent of Internet systems to a halt in November 1988. CERT is located in Pittsburgh, Pennsylvania, at the Software Engineering Institute (SEI), a federally funded research and development center sponsored by the U.S. Department of Defense.

The initial focus of CERT was incident response and analysis. Incidents include successful attacks such as compromises and denials of service, as well as attack attempts, probes, and scans. Since 1988, CERT has received more than 22,665 hotline calls reporting computer security incidents or requesting information and has handled more than 319,992 computer security incidents. The number of incidents reported each year continues to grow.

Responding to incidents, while necessary, is insufficient to secure the Internet and interconnected information systems. Analysis indicates that the majority of incidents is caused by trojans, social engineering, and the exploitation of software vulnerabilities, including software defects, design decisions, configuration decisions, and unexpected interactions among systems. CERT monitors public sources of vulnerability information and regularly receives reports of vulnerabilities. Since 1995, more than 16,726 vulnerabilities have been reported. When a report is received, CERT analyzes the potential vulnerability and works with technology producers to inform them of security deficiencies in their products and to facilitate and track their responses to those problems.[1]

1. CERT interacts with more than 1,900 hardware and software developers.

Similar to incident reports, vulnerability reports continue to grow at an alarming rate.[2] While managing vulnerabilities pushes the process upstream, it is again insufficient to address the issues of Internet and information system security. To address the growing number of both vulnerabilities and incidents, it is increasingly apparent that the problem must be attacked at the source by working to prevent the introduction of software vulnerabilities during software development and ongoing maintenance. Analysis of existing vulnerabilities indicates that a relatively small number of root causes accounts for the majority of vulnerabilities. *The goal of this book is to educate developers about these root causes and the steps that can be taken so that vulnerabilities are not introduced.*

Audience

Secure Coding in C and C++ should be useful to anyone involved in the development or maintenance of software in C and C++.

- If you are a *C/C++ programmer*, this book will teach you how to identify common programming errors that result in software vulnerabilities, understand how these errors are exploited, and implement a solution in a secure fashion.
- If you are a *software project manager*, this book identifies the risks and consequences of software vulnerabilities to guide investments in developing secure software.
- If you are a *computer science student*, this book will teach you programming practices that will help you to avoid developing bad habits and enable you to develop secure programs during your professional career.
- If you are a *security analyst*, this book provides a detailed description of common vulnerabilities, identifies ways to detect these vulnerabilities, and offers practical avoidance strategies.

Organization and Content

Secure Coding in C and C++ provides practical guidance on secure practices in C and C++ programming. Producing secure programs requires secure designs.

2. See www.cert.org/stats/cert_stats.html for current statistics.

However, even the best designs can lead to insecure programs if developers are unaware of the many security pitfalls inherent in C and C++ programming. This book provides a detailed explanation of common programming errors in C and C++ and describes how these errors can lead to code that is vulnerable to exploitation. The book concentrates on security issues intrinsic to the C and C++ programming languages and associated libraries. It does *not* emphasize security issues involving interactions with external systems such as databases and Web servers, as these are rich topics on their own. The intent is that this book be useful to anyone involved in developing secure C and C++ programs regardless of the specific application.

Secure Coding in C and C++ is organized around functional capabilities commonly implemented by software engineers that have potential security consequences, such as formatted output and arithmetic operations. Each chapter describes insecure programming practices and common errors that can lead to vulnerabilities, how these programming flaws can be exploited, the potential consequences of exploitation, and secure alternatives. Root causes of software vulnerabilities, such as buffer overflows, integer type range errors, and invalid format strings, are identified and explained where applicable. Strategies for securely implementing functional capabilities are described in each chapter, as well as techniques for discovering vulnerabilities in existing code.

This book contains the following chapters:

- **Chapter 1** provides an overview of the problem, introduces security terms and concepts, and provides insight into why so many vulnerabilities are found in C and C++ programs.
- **Chapter 2** describes string manipulation in C and C++, common security flaws, and resulting vulnerabilities, including buffer overflow and stack smashing. Both code and arc injection exploits are examined.
- **Chapter 3** introduces *arbitrary memory write* exploits that allow an attacker to write a single address to any location in memory. This chapter describes how these exploits can be used to execute arbitrary code on a compromised machine. Vulnerabilities resulting from arbitrary memory writes are discussed in later chapters.
- **Chapter 4** describes dynamic memory management. Dynamically allocated buffer overflows, writing to freed memory, and double-free vulnerabilities are described.
- **Chapter 5** covers integral security issues (security issues dealing with integers), including integer overflows, sign errors, and truncation errors.

- **Chapter 6** describes the correct and incorrect use of formatted output functions. Both format string and buffer overflow vulnerabilities resulting from the incorrect use of these functions are described.
- **Chapter 7** focuses on concurrency and vulnerabilities that can result from deadlock, race conditions, and invalid memory access sequences.
- **Chapter 8** describes common vulnerabilities associated with file I/O, including race conditions and time of check, time of use (TOCTOU) vulnerabilities.
- **Chapter 9** recommends specific development practices for improving the overall security of your C / C++ application. These recommendations are in addition to the recommendations included in each chapter for addressing specific vulnerability classes.

Secure Coding in C and C++ contains hundreds of examples of secure and insecure code as well as sample exploits. Almost all of these examples are in C and C++, although comparisons are drawn with other languages. The examples are implemented for Windows and Linux operating systems. While the specific examples typically have been compiled and tested in one or more specific environments, vulnerabilities are evaluated to determine whether they are specific to or generalizable across compiler version, operating system, microprocessor, applicable C or C++ standards, little or big endian architectures, and execution stack architecture.

This book, as well as the online course based on it, focuses on common programming errors using C and C++ that frequently result in software vulnerabilities. However, because of size and space constraints, not every potential source of vulnerabilities is covered. Additional and updated information, event schedules, and news related to *Secure Coding in C and C++* are available at www.cert.org/books/secure-coding/. Vulnerabilities discussed in the book are also cross-referenced with real-world examples from the US-CERT Vulnerability Notes Database at www.kb.cert.org/vuls/.

Access to the online secure coding course that accompanies this book is available through Carnegie Mellon's Open Learning Initiative (OLI) at https://oli.cmu.edu/. Enter the course key: 0321822137.

Acknowledgments

I would like to acknowledge the contributions of all those who made this book possible. First, I would like to thank Noopur Davis, Chad Dougherty, Doug Gwyn, David Keaton, Fred Long, Nancy Mead, Robert Mead, Gerhard Muenz, Rob Murawski, Daniel Plakosh, Jason Rafail, David Riley, Martin Sebor, and David Svoboda for contributing chapters to this book. I would also like to thank the following researchers for their contributions: Omar Alhazmi, Archie Andrews, Matthew Conover, Jeffrey S. Gennari, Oded Horovitz, Poul-Henning Kamp, Doug Lea, Yashwant Malaiya, John Robert, and Tim Wilson.

I would also like to thank SEI and CERT managers who encouraged and supported my efforts: Jeffrey Carpenter, Jeffrey Havrilla, Shawn Hernan, Rich Pethia, and Bill Wilson.

Thanks also to my editor, Peter Gordon, and to the folks at Addison-Wesley: Jennifer Andrews, Kim Boedigheimer, John Fuller, Eric Garulay, Stephane Nakib, Elizabeth Ryan, and Barbara Wood.

I would also like to thank everyone who helped develop the Open Learning Initiative course, including the learning scientist who helped design the course, Marsha Lovett, and everyone who helped implement the course, including Norman Bier and Alexandra Drozd.

I would also like to thank the following reviewers for their thoughtful comments and insights: Tad Anderson, John Benito, William Bulley, Corey Cohen, Will Dormann, William Fithen, Robin Eric Fredericksen, Michael Howard, Michael Kaelbling, Amit Kalani, John Lambert, Jeffrey Lanza, David LeBlanc,

Ken MacInnis, Gary McGraw, Randy Meyers, Philip Miller, Patrick Mueller, Dave Mundie, Craig Partridge, Brad Rubbo, Tim Shimeall, Michael Wang, and Katie Washok.

I would like to thank the remainder of the CERT team for their support and assistance, without which I would never have been able to complete this book. And last but not least, I would like to thank our in-house editors and librarians who helped make this work possible: Rachel Callison, Pamela Curtis, Len Estrin, Eric Hayes, Carol J. Lallier, Karen Riley, Sheila Rosenthal, Pennie Walters, and Barbara White.

About the Author

Robert C. Seacord is the Secure Coding Technical Manager in the CERT Program of Carnegie Mellon's Software Engineering Institute (SEI) in Pittsburgh, Pennsylvania. The CERT Program is a trusted provider of operationally relevant cybersecurity research and innovative and timely responses to our nation's cybersecurity challenges. The Secure Coding Initiative works with software developers and software development organizations to eliminate vulnerabilities resulting from coding errors before they are deployed. Robert is also an adjunct professor in the School of Computer Science and the Information Networking Institute at Carnegie Mellon University. He is the author of *The CERT C Secure Coding Standard* (Addison-Wesley, 2008) and coauthor of *Building Systems from Commercial Components* (Addison-Wesley, 2002), *Modernizing Legacy Systems* (Addison-Wesley, 2003), and *The CERT Oracle Secure Coding Standard for Java* (Addison-Wesley, 2011). He has also published more than forty papers on software security, component-based software engineering, Web-based system design, legacy-system modernization, component repositories and search engines, and user interface design and development. Robert has been teaching Secure Coding in C and C++ to private industry, academia, and government since 2005. He started programming professionally for IBM

in 1982, working in communications and operating system software, processor development, and software engineering. Robert has also worked at the X Consortium, where he developed and maintained code for the Common Desktop Environment and the X Window System. He represents Carnegie Mellon University (CMU) at the ISO/IEC JTC1/SC22/WG14 international standardization working group for the C programming language.

Current and former members of the CERT staff who contributed to the development of this book. From left to right: Daniel Plakosh, Archie Andrews, David Svoboda, Dean Sutherland, Brad Rubbo, Jason Rafail, Robert Seacord, Chad Dougherty.

Chapter 1

Running with Scissors

To live without evil belongs
only to the gods.
—Sophocles, *Fragments*, l. 683

Computer systems are not vulnerable to attack. *We* are vulnerable to attack through our computer systems.

The W32.Blaster.Worm, discovered "in the wild" on August 11, 2003, is a good example of how security flaws in software make us vulnerable. Blaster can infect any unpatched system connected to the Internet without user involvement. Data from Microsoft suggests that at least 8 million Windows systems have been infected by this worm [Lemos 2004]. Blaster caused a major disruption as some users were unable to use their machines, local networks were saturated, and infected users had to remove the worm and update their machines.

The chronology, shown in Figure 1.1, leading up to and following the criminal launch of the Blaster worm shows the complex interplay among software companies, security researchers, persons who publish exploit code, and malicious attackers.

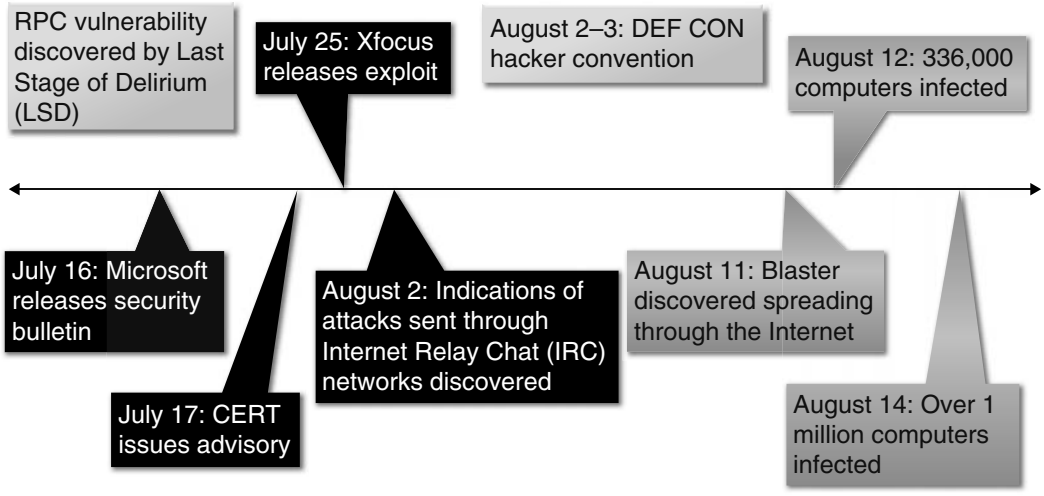

Figure 1.1 Blaster timeline

The Last Stage of Delirium (LSD) Research Group discovered a buffer overflow vulnerability in RPC[1] that deals with message exchange over TCP/IP. The failure is caused by incorrect handling of malformed messages. The vulnerability affects a distributed component object model (DCOM) interface with RPC that listens on RPC-enabled ports. This interface handles object activation requests sent by client machines to the server. Successful exploitation of this vulnerability allows an attacker to run arbitrary code with local system privileges on an affected system.

In this case, the LSD group followed a policy of responsible disclosure by working with the vendor to resolve the issue before going public. On July 16, 2003, Microsoft released Microsoft Security Bulletin MS03-026,[2] LSD released a special report, and the coordination center at CERT (CERT/CC) released vulnerability note VU#568148[3] detailing this vulnerability and providing patch and workaround information. On the following day, the CERT/CC also issued CERT Advisory CA-2003-16, "Buffer Overflow in Microsoft RPC."[4]

1. Remote procedure call (RPC) is an interprocess communication mechanism that allows a program running on one computer to execute code on a remote system. The Microsoft implementation is based on the Open Software Foundation (OSF) RPC protocol but adds Microsoft-specific extensions.
2. See www.microsoft.com/technet/security/bulletin/MS03-026.mspx.
3. See www.kb.cert.org/vuls/id/568148.
4. See www.cert.org/advisories/CA-2003-16.html.

Nine days later, on July 25, a security research group called Xfocus published an exploit for the vulnerability identified by the security bulletin and patch. Xfocus describes itself as "a non-profit and free technology organization" that was founded in 1998 in China and is devoted to "research and demonstration of weaknesses related to network services and communication security." In essence, Xfocus analyzed the Microsoft patch by reverse engineering it to identify the vulnerability, developed a means to attack the vulnerability, and made the exploit publicly available [Charney 2003].

H. D. Moore (founder of the Metasploit Project) improved the Xfocus code to exploit additional operating systems. Soon exploit tools were released that enabled hackers to send commands through IRC networks. Indications of these attacks were discovered on August 2 [de Kere 2003].

With the DEF CON hacker convention scheduled for August 2–3, it was widely expected that a worm would be released that used this exploit (not necessarily by people attending DEF CON but simply because of the attention to hacking that the conference brings). The Department of Homeland Security issued an alert on August 1, and the Federal Computer Incident Response Center (FedCIRC), the National Communications System (NCS), and the National Infrastructure Protection Center (NIPC) were actively monitoring for exploits. On August 11, only 26 days after release of the patch, the Blaster worm was discovered as it spread through the Internet. Within 24 hours, Blaster had infected 336,000 computers [Pethia 2003a]. By August 14, Blaster had infected more than 1 million computers; at its peak, it was infecting 100,000 systems per hour [de Kere 2003].

Blaster is an aggressive worm that propagates via TCP/IP, exploiting a vulnerability in the DCOM RPC interface of Windows. When Blaster executes, it checks to see if the computer is already infected and if the worm is running. If so, the worm does not infect the computer a second time. Otherwise, Blaster adds the value

```
"windows auto update"="msblast.exe"
```

to the registry key

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

so that the worm runs when Windows is started. Next, Blaster generates a random IP address and attempts to infect the computer with that address. The worm sends data on TCP port 135 to exploit the DCOM RPC vulnerability on either Windows XP or Windows 2000. Blaster listens on UDP port 69 for a request from a computer to which it was able to connect using the DCOM

RPC exploit. When it receives a request, it sends the `msblast.exe` file to that computer and executes the worm [Hoogstraten 2003].

The worm uses `cmd.exe` to create a back-door remote shell process that listens on TCP port 4444, allowing an attacker to issue remote commands on the compromised system. Blaster also attempts to launch a denial-of-service (DoS) attack on Windows Update to prevent users from downloading the patch. The DoS attack is launched on a particular date in the form of a SYN flood[5] on port 80 of `windowsupdate.com`.

Even when Blaster does not successfully infect a target system, the DCOM RPC buffer overflow exploit kills the `svchost.exe` process on Windows NT, Windows 2000, Windows XP, and Windows 2003 systems scanned by the worm. On Windows NT and Windows 2000, the system becomes unstable and hangs. Windows XP and Windows 2003 initiate a reboot by default.

The launch of Blaster was not a surprise. On June 25, 2003, a month before the initial vulnerability disclosure that led to Blaster, Richard Pethia, director of the CERT/CC, testified before the House Select Committee on Homeland Security Subcommittee on Cybersecurity, Science, and Research and Development [Pethia 2003a] that

> the current state of Internet security is cause for concern. Vulnerabilities associated with the Internet put users at risk. Security measures that were appropriate for mainframe computers and small, well-defined networks inside an organization are not effective for the Internet, a complex, dynamic world of interconnected networks with no clear boundaries and no central control. Security issues are often not well understood and are rarely given high priority by many software developers, vendors, network managers, or consumers.

Economic damage from the Blaster worm has been estimated to be at least $525 million. The cost estimates include lost productivity, wasted hours, lost sales, and extra bandwidth costs [Pethia 2003b]. Although the impact of Blaster was impressive, the worm could easily have been more damaging if, for example, it erased files on infected systems. Based on a parameterized worst-case analysis using a simple damage model, Nicholas Weaver and Vern Paxson [Weaver 2004] estimate that a plausible worst-case worm could cause $50 billion or more in direct economic damage by attacking widely used services in Microsoft Windows and carrying a highly destructive payload (for example, destroying the primary hard drive controller, overwriting CMOS RAM, or erasing flash memory).

5. SYN flooding is a method that the user of a hostile client program can use to conduct a DoS attack on a computer server. The hostile client repeatedly sends SYN (synchronization) packets to every port on the server, using fake IP addresses.

```
01     error_status_t _RemoteActivation(
02          ..., WCHAR *pwszObjectName, ... ) {
03       *phr = GetServerPath(pwszObjectName, &pwszObjectName);
04      ...
05     }
06
07     HRESULT GetServerPath(
08          WCHAR *pwszPath, WCHAR **pwszServerPath ){
09       WCHAR *pwszFinalPath = pwszPath;
10       WCHAR wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1];
11       hr = GetMachineName(pwszPath, wszMachineName);
12       *pwszServerPath = pwszFinalPath;
13     }
14
15     HRESULT GetMachineName(
16       WCHAR *pwszPath,
17       WCHAR wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1])
18     {
19       pwszServerName = wszMachineName;
20       LPWSTR pwszTemp = pwszPath + 2;
21       while ( *pwszTemp != L'\\' )
22         *pwszServerName++ = *pwszTemp++;
23        ...
24     }
```

Figure 1.2 Flawed logic exploited by the W32.Blaster.Worm

The flawed logic exploited by the W32.Blaster.Worm is shown in Figure 1.2.[6] The error is that the `while` loop on lines 21 and 22 (used to extract the host name from a longer string) is not sufficiently bounded. Once identified, this problem can be trivially repaired, for example, by adding a second condition to the controlling expression of the `while` loop, which terminates the search before the bounds of the wide string referenced by `pwszTemp` or by `pwszServerName` is exceeded.

1.1 Gauging the Threat

The risk of producing insecure software systems can be evaluated by looking at historic risk and the potential for future attacks. Historic risk can be measured by looking at the type and cost of perpetrated crimes, although it

6. Special thanks to Microsoft for supplying this code fragment.

is generally believed that these crimes are underreported. The potential for future attacks can be at least partially gauged by evaluating emerging threats and the security of existing software systems.

What Is the Cost?

The *2010 CyberSecurity Watch Survey*, conducted by *CSO* magazine in cooperation with the U.S. Secret Service, the Software Engineering Institute CERT Program at Carnegie Mellon University, and Deloitte's Center for Security and Privacy Solutions [CSO 2010], revealed a decrease in the number of cybercrime victims between 2007 and 2009 (60 percent versus 66 percent) but a significant increase in the number of cybercrime incidents among the affected organizations. Between August 2008 and July 2009, more than one-third (37 percent) of the survey's 523 respondents experienced an increase in cybercrimes compared to the previous year, and 16 percent reported an increase in monetary losses. Of those who experienced e-crimes, 25 percent reported operational losses, 13 percent stated financial losses, and 15 percent declared harm to reputation as a result. Respondents reported an average loss of $394,700 per organization because of e-crimes.

Estimates of the costs of cybercrime in the United States alone range from millions to as high as a trillion dollars a year. The true costs, however, are hard to quantify for a number of reasons, including the following:

- A high number of cybercrimes (72 percent, according to the *2010 CyberSecurity Watch Survey* [CSO 2010]) go unreported or even unnoticed.
- Statistics are sometimes unreliable—either over- or underreported, depending on the source (a bank, for example, may be motivated to underreport costs to avoid loss of consumer trust in online banking). According to the *2010/2011 Computer Crime and Security Survey* conducted by the Computer Security Institute (CSI), "Fewer respondents than ever are willing to share specific information about dollar losses they incurred" [CSI 2011].
- Indirect costs, such as loss of consumer trust in online services (which for a bank, for example, leads to reduced revenues from electronic transaction fees and higher costs for maintaining staff [Anderson 2012]), are also over- or underreported or not factored in by all reporting agencies.
- The lines are often blurred between traditional crimes (such as tax and welfare fraud that today are considered cybercrimes only because a large part of these interactions are now conducted online) and new crimes that "owe their existence to the Internet" [Anderson 2012].

Ross Anderson and his colleagues conducted a systematic study of the costs of cybercrime in response to a request from the UK Ministry of Defence [Anderson 2012]. Table 1.1 highlights some of their findings on the estimated global costs of cybercrime.

Table 1.1 Judgment on Coverage of Cost Categories by Known Estimates*

Type of Cybercrime	Global Estimate ($ million)	Reference Period
Cost of Genuine Cybercrime		
Online banking fraud	320	2007
Phishing	70	2010
Malware (consumer)	300	2010
Malware (businesses)	1,000	2010
Bank technology countermeasures	97	2008–10
Fake antivirus	22	2010
Copyright-infringing software	150	2011
Copyright-infringing music, etc.	288	2010
Patent-infringing pharmaceutical	10	2011
Stranded traveler scam	200	2011
Fake escrow scam	1,000[a]	2011
Advance-fee fraud		2011
Cost of Transitional Cybercrime		
Online payment card fraud	4,200[a]	2010
Offline payment card fraud		
Domestic	2,100[a]	2010
International	2,940[a]	2010
Bank/merchant defense costs	2,400	2010
Indirect costs of payment fraud		
Loss of confidence (consumers)	10,000[a]	2010
Loss of confidence (merchants)	20,000[a]	2009
PABX fraud	4,960	2011

continues

Table 1.1 Judgment on Coverage of Cost Categories by Known Estimates*(*continued*)

Type of Cybercrime	Global Estimate ($ million)	Reference Period
Cost of Cybercriminal Infrastructure		
Expenditure on antivirus	3,400[a]	2012
Cost to industry of patching	1,000	2010
ISP cleanup expenditures	40[a]	2010
Cost to users of cleanup	10,000[a]	2012
Defense costs of firms generally	10,000	2010
Expenditure on law enforcement	400[a]	2010
Cost of Traditional Crimes Becoming "Cyber"		
Welfare fraud	20,000[a]	2011
Tax fraud	125,000[a]	2011
Tax filing fraud	5,200	2011

*Source: Adapted from R. Anderson et al., "Measuring the Cost of Cybercrime," paper presented at the 11th Annual Workshop on the Economics of Information Security, 2012. http://weis2012.econinfosec.org/papers/Anderson_WEIS2012.pdf.
[a] Estimate is scaled using UK data and based on the United Kingdom's share of world GDP (5 percent); extrapolations from UK numbers to the global scale should be interpreted with utmost caution.

Who Is the Threat?

The term *threat* has many meanings in computer security. One definition (often used by the military) is a person, group, organization, or foreign power that has been the source of past attacks or may be the source of future attacks. Examples of possible threats include hackers, insiders, criminals, competitive intelligence professionals, terrorists, and information warriors.

Hackers. Hackers include a broad range of individuals of varying technical abilities and attitudes. Hackers often have an antagonistic response to authority and often exhibit behaviors that appear threatening [Thomas 2002]. Hackers are motivated by curiosity and peer recognition from other hackers. Many hackers write programs that *expose vulnerabilities* in computer software. The methods these hackers use to disclose vulnerabilities vary from a policy

of responsible disclosure[7] to a policy of full disclosure (telling everything to everyone as soon as possible). As a result, hackers can be both a benefit and a bane to security. Hackers whose primary intent is to gain unauthorized access to computer systems to steal or corrupt data are often referred to as *crackers*.

Insiders. The insider threat comes from a current or former employee or contractor of an organization who has legitimate access to the information system, network, or data that was compromised [Andersen 2004]. Because insiders have legitimate access to their organization's networks and systems, they do not need to be technically sophisticated to carry out attacks. The threat increases with technically sophisticated insiders, who can launch attacks with immediate and widespread impact. These technical insiders may also know how to cover their tracks, making it more difficult to discover their identities. Insiders can be motivated by a variety of factors. Financial gain is a common motive in certain industries, and revenge can span industries. Theft of intellectual property is prevalent for financial gain or to enhance an employee's reputation with a new employer. Since 2001, the CERT Insider Threat Center has collected and analyzed information about more than 700 insider cybercrimes, ranging from national security espionage to theft of trade secrets [Cappelli 2012].

Criminals. Criminals are individuals or members of organized crime syndicates who hope to profit from their activities. Common crimes include auction fraud and identity theft. Phishing attacks that use spoofed e-mails and fraudulent Web sites designed to fool recipients into divulging personal financial data such as credit card numbers, account user names and passwords, and Social Security numbers have increased in number and sophistication. Cybercriminals may also attempt to break into systems to retrieve credit card information (from which they can profit directly) or sensitive information that can be sold or used for blackmail.

Competitive Intelligence Professionals. Corporate spies call themselves *competitive intelligence professionals* and even have their own professional association.[8] Competitive intelligence professionals may work from inside a target organization, obtaining employment to steal and market trade secrets or conduct other forms of corporate espionage. Others may gain access through the Internet, dial-up lines, physical break-ins, or from partner (vendor, customer,

7. The CERT/CC Vulnerability Disclosure Policy is available at www.cert.org/kb/vul_disclosure.html.

8. See www.scip.org.

or reseller) networks that are linked to another company's network. Since the end of the Cold War, a number of countries have been using their intelligence-gathering capabilities to obtain proprietary information from major corporations.

Terrorists. Cyberterrorism is unlawful attacks or threats of attack against computers, networks, and other information systems to intimidate or coerce a government or its people to further a political or social objective [Denning 2000]. Because terrorists have a different objective from, say, criminals, the attacks they are likely to execute are different. For example, terrorists may be interested in attacking critical infrastructure such as a supervisory control and data acquisition (SCADA) system, which controls devices that provide essential services such as power or gas. While this is a concern, these systems are considerably more difficult to attack than typical corporate information systems. Politically motivated cyber attacks, as a form of protest, usually involve Web site defacements (with a political message) or some type of DoS attack and are usually conducted by loosely organized hacker groups (such as Anonymous) or individuals with hacker skills who are sympathetic to a particular cause or who align themselves with a particular side in a conflict [Kerr 2004].

Information Warriors. The United States faces a "long-term challenge in cyberspace from foreign intelligence agencies and militaries," according to the Center for Strategic and International Studies (CSIS). Intrusions by unknown foreign entities have been reported by the departments of Defense, Commerce, Homeland Security, and other government agencies [CSIS 2008]. The CSIS maintains a list of significant cyber events,[9] which tracks reported attacks internationally. NASA's inspector general, for example, reported that 13 APT (advanced persistent threat) attacks successfully compromised NASA computers in 2011. In one attack, intruders stole the credentials of 150 users that could be used to gain unauthorized access to NASA systems. And in December 2011, it was reported that the U.S. Chamber of Commerce computer networks had been completely penetrated for more than a year by hackers with ties to the People's Liberation Army. The hackers had access to everything in Chamber computers, including member company communications and industry positions on U.S. trade policy. Information warriors have successfully accessed critical military technologies and valuable intellectual property, and they pose a serious, ongoing threat to the U.S. economy and national security.

9. See http://csis.org/publication/cyber-events-2006.

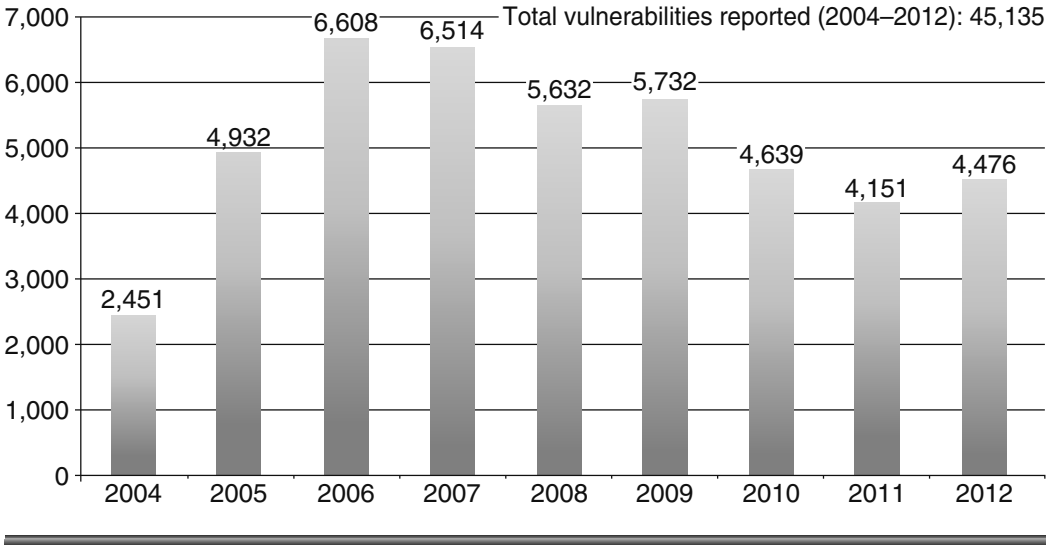

Figure 1.3 Vulnerabilities cataloged in the NVD

Software Security

The CERT/CC monitors public sources of vulnerability information and regularly receives reports of vulnerabilities. Vulnerability information is published as CERT vulnerability notes and as US-CERT vulnerability notes.[10] The CERT/CC is no longer the sole source of vulnerability reports; many other organizations, including Symantec and MITRE, also report vulnerability data.

Currently, one of the best sources of vulnerabilities is the National Vulnerability Database (NVD) of the National Institute of Standards and Technology (NIST). The NVD consolidates vulnerability information from multiple sources, including the CERT/CC, and consequently contains a superset of vulnerabilities from its various feeds.

Figure 1.3 shows the number of vulnerabilities cataloged in the NVD from 2004 through the third quarter of 2012. The first edition of this book charted vulnerabilities reported to the CERT/CC from 1995 through 2004. Unfortunately, these numbers have only continued to climb.

Dr. Gregory E. Shannon, Chief Scientist for CERT, characterized the software security environment in his testimony before the House Committee on Homeland Security [Shannon 2011]:

10. See www.kb.cert.org/vuls.

> Today's operational cyber environments are complex and dynamic. User needs and environmental factors are constantly changing, which leads to unanticipated usage, reconfiguration, and continuous evolution of practices and technologies. New defects and vulnerabilities in these environments are continually being discovered, and the means to exploit these environments continues to rise. The CERT Coordination Center cataloged ~250,000 instances of malicious artifacts last month alone. From this milieu, public and private institutions respond daily to repeated attacks and also to the more serious previously un-experienced failures (but not necessarily unexpected); both demand rapid, capable and agile responses.

Because the number and sophistication of threats are increasing faster than our ability to develop and deploy more secure systems, the risk of future attacks is considerable and increasing.

1.2 Security Concepts

Computer security prevents attackers from achieving objectives through unauthorized access or unauthorized use of computers and networks [Howard 1997]. Security has *developmental* and *operational* elements. Developing secure code requires secure designs and flawless implementations. Operational security requires securing deployed systems and networks from attack. Which comes first is a chicken-and-egg problem; both are practical necessities. Even if perfectly secure software can be developed, it still needs to be deployed and configured in a secure fashion. Even the most secure vault ever designed, for example, is vulnerable to attack if the door is left open. This situation is further exacerbated by end user demands that software be easy to use, configure, and maintain while remaining inexpensive.

Figure 1.4 shows the relationships among these security concepts.

Programs are constructed from software components and custom-developed source code. *Software components* are the elements from which larger software programs are composed [Wallnau 2002]. Software components include shared libraries such as dynamic-link libraries (DLLs), ActiveX controls, Enterprise JavaBeans, and other compositional units. Software components may be linked into a program or dynamically bound at runtime. Software components, however, are not directly executed by an end user, except as part of a larger program. Therefore, software components cannot have vulnerabilities because they are not executable outside of the context of a program. *Source code* comprises program instructions in their original form. The word *source* differentiates code from various other forms that code can have (for example,

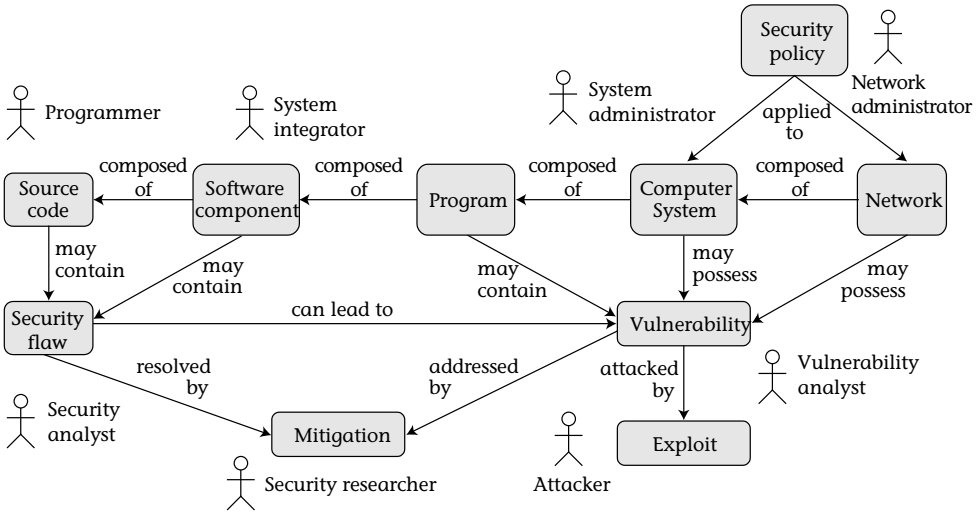

Figure 1.4 Security concepts, actors, and relationships

object code and executable code). Although it is sometimes necessary or desirable to analyze code in these other, nonsource forms (for example, when integrating components from third parties where source code is unavailable), we focus on source code because a principal audience for this book is software developers (who normally have access to the source code).

Figure 1.4 also shows relationships among actors and artifacts. These roles vary among organizations, but the following definitions are used in this book:

- A *programmer* is concerned with properties of source code such as correctness, performance, and security.
- A *system integrator* is responsible for integrating new and existing software components to create programs or systems that satisfy a particular set of customer requirements.
- *System administrators* are responsible for managing and securing one or more systems, including installing and removing software, installing patches, and managing system privileges.
- *Network administrators* are responsible for managing the secure operations of networks.
- A *security analyst* is concerned with properties of security flaws and how to identify them.
- A *vulnerability analyst* is concerned with analyzing vulnerabilities in existing and deployed programs.

- A *security researcher* develops mitigation strategies and solutions and may be employed in industry, academia, or government.
- The *attacker* is a malicious actor who exploits vulnerabilities to achieve an objective. These objectives vary depending on the threat. The attacker can also be referred to as the adversary, malicious user, hacker, or other alias.

Security Policy

A security policy is a set of rules and practices that are normally applied by system and network administrators to their systems to secure them from threats. The following definition is taken verbatim from RFC 2828, the *Internet Security Glossary* [Internet Society 2000]:

> **Security Policy**
> A set of rules and practices that specify or regulate how a system or organization provides security services to protect sensitive and critical system resources.

Security policies can be both implicit and explicit. Security policies that are documented, well known, and visibly enforced can help establish expected user behavior. However, the lack of an explicit security policy does not mean an organization is immune to attack because it has no security policy to violate.

Security Flaws

Software engineering has long been concerned with the elimination of *software defects*. A software defect is the encoding of a human error into the software, including omissions. Software defects can originate at any point in the software development life cycle. For example, a defect in a deployed product can originate from a misstated or misrepresented requirement.

> **Security Flaw**
> A software defect that poses a potential security risk.

Not all software defects pose a security risk. Those that do are *security flaws*. If we accept that a security flaw is a software defect, then we must also accept that by eliminating all software defects, we can eliminate all security flaws.

This premise underlies the relationship between software engineering and secure programming. An increase in quality, as might be measured by defects per thousand lines of code, would likely also result in an increase in security. Consequently, many tools, techniques, and processes that are designed to eliminate software defects also can be used to eliminate security flaws.

However, many security flaws go undetected because traditional software development processes seldom assume the existence of attackers. For example, testing will normally validate that an application behaves correctly for a *reasonable* range of user inputs. Unfortunately, attackers are seldom reasonable and will spend an inordinate amount of time devising inputs that will break a system. To *identify* and *prioritize* security flaws according to the risk they pose, existing tools and methods must be extended or supplemented to assume the existence of an attacker.

Vulnerabilities

Not all security flaws lead to vulnerabilities. However, a security flaw can cause a program to be vulnerable to attack when the program's input data (for example, command-line parameters) crosses a security boundary en route to the program. This may occur when a program containing a security flaw is installed with execution privileges greater than those of the person running the program or is used by a network service where the program's input data arrives via the network connection.

Vulnerability

A set of conditions that allows an attacker to violate an explicit or implicit security policy.

This same definition is used in the draft ISO/IEC TS 17961 *C Secure Coding Rules* technical specification [Seacord 2012a]. A security flaw can also exist without all the preconditions required to create a vulnerability. For example, a program can contain a defect that allows a user to run arbitrary code inheriting the permissions of that program. This is not a vulnerability if the program has no special permissions and can be accessed only by local users, because there is no possibility that a security policy will be violated. However, this defect is still a security flaw in that the program may be redeployed or reused in a system in which privilege escalation may occur, allowing an attacker to execute code with elevated privileges.

Vulnerabilities can exist without a security flaw. Because security is a quality attribute that must be traded off with other quality attributes such as

performance and usability [Bass 2013], software designers may *intentionally choose* to leave their product vulnerable to some form of exploitation. Making an intentional decision not to eliminate a vulnerability does not mean the software is secure, only that the software designer has accepted the risk on behalf of the software consumer.

Figure 1.4 shows that programs may *contain* vulnerabilities, whereas computer systems and networks may *possess* them. This distinction may be viewed as minor, but programs are not actually vulnerable until they are operationally deployed on a computer system or network. No one can attack you using a program that is on a disk in your office if that program is not installed—and installed in such a way that an attacker can exploit it to violate a *security policy*. Additionally, real-world vulnerabilities are often determined by a specific configuration of that software that enables an innate security flaw to be exploited. Because this distinction is somewhat difficult to communicate, we often talk about programs *containing vulnerabilities* or *being vulnerable* both in this book and in CERT/CC vulnerability notes and advisories.

Exploits

Vulnerabilities in software are subject to exploitation. Exploits can take many forms, including worms, viruses, and trojans.

Exploit
A technique that takes advantage of a security vulnerability to violate an explicit or implicit security policy.

The existence of exploits makes security analysts nervous. Therefore, fine distinctions are made regarding the purpose of exploit code. For example, proof-of-concept exploits are developed to prove the existence of a vulnerability. Proof-of-concept exploits may be necessary, for example, when vendors are reluctant to admit to the existence of a vulnerability because of negative publicity or the cost of providing patches. Vulnerabilities can also be complex, and often a proof-of-concept exploit is necessary to prove to vendors that a vulnerability exists.

Proof-of-concept exploits are beneficial when properly managed. However, it is readily apparent how a proof-of-concept exploit in the wrong hands can be quickly transformed into a worm or virus or used in an attack.

Security researchers like to distinguish among different types of exploits, but the truth is, of course, that all forms of exploits encode knowledge, and knowledge is power. Understanding how programs can be exploited is a valuable tool that can be used to develop secure software. However, disseminating

exploit code against known vulnerabilities can be damaging to everyone. In writing this book, we decided to include only sample exploits for sample programs. While this information can be used for multiple purposes, significant knowledge and expertise are still required to create an actual exploit.

Mitigations

A mitigation is a *solution* for a software flaw or a workaround that can be applied to prevent exploitation of a vulnerability.[11] At the source code level, mitigations can be as simple as replacing an unbounded string copy operation with a bounded one. At a system or network level, a mitigation might involve turning off a port or filtering traffic to prevent an attacker from accessing a vulnerability.

The preferred way to eliminate security flaws is to find and correct the actual defect. However, in some cases it can be more cost-effective to eliminate the security flaw by preventing malicious inputs from reaching the defect. Generally, this approach is less desirable because it requires the developer to understand and protect the code against all manner of attacks as well as to identify and protect all paths in the code that lead to the defect.

Mitigation

Methods, techniques, processes, tools, or runtime libraries that can prevent or limit exploits against vulnerabilities.

Vulnerabilities can also be *addressed* operationally by isolating the vulnerability or preventing malicious inputs from reaching the vulnerable code. Of course, operationally addressing vulnerabilities significantly increases the cost of mitigation because the cost is pushed out from the developer to system administrators and end users. Additionally, because the mitigation must be successfully implemented by host system administrators or users, there is increased risk that vulnerabilities will not be properly addressed in all cases.

1.3 C and C++

The decision to write a book on secure programming in C and C++ was based on the popularity of these languages, the enormous legacy code base, and the amount of new code being developed in these languages. The TIOBE index

11. Mitigations are alternatively called *countermeasures* or *avoidance strategies*.

is one measure of the popularity of programming languages. Table 1.2 shows the TIOBE Index for January 2013, and Table 1.3 shows long-term trends in language popularity.

Additionally, the vast majority of vulnerabilities that have been reported to the CERT/CC have occurred in programs written in either C or C++. Before examining why, we look briefly at the history of these languages.

Table 1.2 TIOBE Index (January 2013)

Position Jan 2013	Position Jan 2012	Programming Language	Ratings Jan 2013	Delta Jan 2012	Status
1	2	C	17.855%	+0.89%	A
2	1	Java	17.417%	-0.05%	A
3	5	Objective-C	10.283%	+3.37%	A
4	4	C++	9.140%	+1.09%	A
5	3	C#	6.196%	-2.57%	A
6	6	PHP	5.546%	-0.16%	A
7	7	(Visual) Basic	4.749%	+0.23%	A
8	8	Python	4.173%	+0.96%	A
9	9	Perl	2.264%	-0.50%	A
10	10	JavaScript	1.976%	-0.34%	A
11	12	Ruby	1.775%	+0.34%	A
12	24	Visual Basic .NET	1.043%	+0.56%	A
13	13	Lisp	0.953%	-0.16%	A
14	14	Pascal	0.932%	+0.14%	A
15	11	Delphi/Object Pascal	0.919%	-0.65%	A
16	17	Ada	0.651%	+0.02%	B
17	23	MATLAB	0.641%	+0.13%	B
18	20	Lua	0.633%	+0.07%	B
19	21	Assembly	0.629%	+0.08%	B
20	72	Bash	0.613%	+0.49%	B

Table 1.3 TIOBE Long Term History (January 2013)

Programming Language	Position Jan 2013	Position Jan 2008	Position Jan 1998	Position Jan 1988
C	1	2	1	1
Java	2	1	4	—
Objective-C	3	45	—	—
C++	4	5	2	7
C#	5	8	—	—
PHP	6	4	—	-
(Visual) Basic	7	3	3	5
Python	8	6	30	—
Perl	9	7	17	—
JavaScript	10	10	26	—
Lisp	13	19	6	2
Ada	16	22	12	3

A Brief History

Dennis Ritchie presented "The Development of the C Language" at the Second History of Programming Languages conference [Bergin 1996]. The C language was created in the early 1970s as a system implementation language for the UNIX operating system. C was derived from the typeless language B [Johnson 1973], which in turn was derived from Basic Combined Programming Language (BCPL) [Richards 1979]. BCPL was designed by Martin Richards in the 1960s and used during the early 1970s on several projects. B can be thought of as C without types or, more accurately, BCPL refined and compressed into 8K bytes of memory.

The C Programming Language, often called "K&R" [Kernighan 1978], was originally published in 1978. Language changes around this time were largely focused on portability as a result of porting code to the Interdata 8/32 computer. At the time, C still showed strong signs of its typeless origins.

ANSI established the X3J11 committee in the summer of 1983. ANSI's goal was "to develop a clear, consistent, and unambiguous Standard for the C

programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environments" [ANSI 1989]. X3J11 produced its report at the end of 1989, and this standard was subsequently accepted by the International Organization for Standardization (ISO) as ISO/IEC 9899-1990. There are no technical differences between these publications, although the sections of the ANSI standard were renumbered and became clauses in the ISO standard. This standard, in both forms, is commonly known as C89, or occasionally as C90 (from the dates of ratification). This first edition of the standard was then amended and corrected by ISO/IEC 9899/COR1:1994, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996. The ISO/IEC 9899/AMD1:1995 amendment is commonly known as AMD1; the amended standard is sometimes known as C95.

This first edition of the standard (and amendments) was subsequently replaced by ISO/IEC 9899:1999 [ISO/IEC 1999]. This version of the C Standard is generally referred to as C99. More recently, the second edition of the standard (and amendments) was replaced by ISO/IEC 9899:2011 [ISO/IEC 2011], commonly referred to as C11.

Descendants of C proper include Concurrent C [Gehani 1989], Objective-C [Cox 1991], C* [Thinking 1990], and especially C++ [Stroustrup 1986]. The C language is also widely used as an intermediate representation (as a portable assembly language) for a wide variety of compilers, both for direct descendants like C++ and independent languages like Modula 3 [Nelson 1991] and Eiffel [Meyer 1988].

Of these descendants of C, C++ has been most widely adopted. C++ was written by Bjarne Stroustrup at Bell Labs during 1983–85. Before 1983, Stroustrup added features to C and formed what he called "C with Classes." The term *C++* was first used in 1983.

C++ was developed significantly after its first release. In particular, *The Annotated C++ Reference Manual* (ARM C++) [Ellis 1990] added exceptions and templates, and ISO C++ added runtime type identification (RTTI), namespaces, and a standard library. The most recent version of the C++ Standard is ISO/IEC 14882:2011, commonly called C++11 [ISO/IEC 14882:2011].

The C and C++ languages continue to evolve today. The C Standard is maintained by the international standardization working group for the programming language C (ISO/IEC JTC1 SC22 WG14). The U.S. position is represented by the INCITS PL22.11 C Technical Committee. The C++ Standard is maintained by the international standardization working group for the programming language C++ (ISO/IEC JTC1 SC22 WG21). The U.S. position is represented by the INCITS PL22.16 C++ Technical Committee.

What Is the Problem with C?

C is a flexible, high-level language that has been used extensively for over 40 years but is the bane of the security community. What are the characteristics of C that make it prone to programming errors that result in security flaws?

The C programming language is intended to be a *lightweight* language with a small footprint. This characteristic of C leads to vulnerabilities when programmers fail to implement required logic because they assume it is handled by C (but it is not). This problem is magnified when programmers are familiar with superficially similar languages such as Java, Pascal, or Ada, leading them to believe that C protects the programmer better than it actually does. These false assumptions have led to programmers failing to prevent writing beyond the boundaries of an array, failing to catch integer overflows and truncations, and calling functions with the wrong number of arguments.

The original charter for C language standardization contains a number of guiding principles. Of these, point 6 provides the most insight into the source of security problems with the language:

> **Point 6:** Keep the spirit of C. Some of the facets of the spirit of C can be summarized in phrases like
>
> (a) Trust the programmer.
>
> (b) Don't prevent the programmer from doing what needs to be done.
>
> (c) Keep the language small and simple.
>
> (d) Provide only one way to do an operation.
>
> (e) Make it fast, even if it is not guaranteed to be portable.

Proverbs (a) and (b) are directly at odds with security and safety. At the Spring 2007 London meeting of WG14, where the C11 charter was discussed, the idea came up that (a) should be revised to "Trust with verification." Point (b) is felt by WG14 to be critical to the continued success of the C language.

The C Standard [ISO/IEC 2011] defines several kinds of behaviors:

> **Locale-specific behavior:** behavior that depends on local conventions of nationality, culture, and language that each implementation documents. An example of locale-specific behavior is whether the `islower()` function returns true for characters other than the 26 lowercase Latin letters.
>
> **Unspecified behavior:** use of an unspecified value, or other behavior where the C Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance. An example of

> unspecified behavior is the order in which the arguments to a function are evaluated.
>
> **Implementation-defined behavior:** unspecified behavior where each implementation documents how the choice is made. An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.
>
> **Undefined behavior:** behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.

Annex J, "Portability issues," enumerates specific examples of these behaviors in the C language.

An *implementation* is a particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment. An implementation is basically synonymous with a compiler command line, including the selected flags or options. Changing any flag or option can result in generating significantly different executables and is consequently viewed as a separate implementation.

The C Standard also explains how undefined behavior is identified:

> If a "shall" or "shall not" requirement that appears outside of a constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this International Standard by the words "undefined behavior" or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe "behavior that is undefined."

Behavior can be classified as undefined by the C standards committee for the following reasons:

- To give the implementor license not to catch certain program errors that are difficult to diagnose
- To avoid defining obscure corner cases that would favor one implementation strategy over another
- To identify areas of possible conforming language extension: the implementor may augment the language by providing a definition of the officially undefined behavior

Conforming implementations can deal with undefined behavior in a variety of fashions, such as ignoring the situation completely, with unpredictable

results; translating or executing the program in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message); or terminating a translation or execution (with the issuance of a diagnostic message).

Undefined behaviors are extremely dangerous because they are not required to be diagnosed by the compiler and because any behavior can occur in the resulting program. Most of the security vulnerabilities described in this book are the result of exploiting undefined behaviors in code.

Another problem with undefined behaviors is compiler optimizations. Because compilers are not obligated to generate code for undefined behavior, these behaviors are candidates for optimization. By assuming that undefined behaviors will not occur, compilers can generate code with better performance characteristics.

Increasingly, compiler writers are taking advantage of undefined behaviors in the C programming languages to improve optimizations. Frequently, these optimizations interfere with the ability of developers to perform cause-effect analysis on their source code, that is, analyzing the dependence of downstream results on prior results. Consequently, these optimizations eliminate causality in software and increase the probability of software faults, defects, and vulnerabilities.

As suggested by the title of Annex J, unspecified, undefined, implementation-defined, and locale-specific behaviors are all *portability* issues. Undefined behaviors are the most problematic, as their behavior can be well defined for one version of a compiler and can change completely for a subsequent version. The C Standard requires that implementations document and define all implementation-defined and locale-specific characteristics and all extensions.

As we can see from the history of the language, portability was not a major goal at the inception of the C programming language but gradually became important as the language was ported to different platforms and eventually became standardized. The current C Standard identifies two levels of portable program: *conforming* and *strictly conforming*.

A *strictly conforming program* uses only those features of the language and library specified by the C Standard. A strictly conforming program can use conditional features provided the use is guarded by an appropriate conditional inclusion preprocessing directive. It cannot produce output dependent on any unspecified, undefined, or implementation-defined behavior and cannot exceed any minimum implementation limit. A *conforming program* is one that is acceptable to a conforming implementation. Strictly conforming programs are intended to be maximally portable among conforming implementations. Conforming programs may depend on nonportable features of a conforming implementation.

Portability requires that logic be encoded at a level of abstraction independent of the underlying machine architecture and transformed or compiled into the underlying representation. Problems arise from an imprecise understanding of the semantics of these abstractions and how they translate into machine-level instructions. This lack of understanding leads to mismatched assumptions, security flaws, and vulnerabilities.

The C programming language lacks *type safety*. Type safety consists of two attributes: *preservation* and *progress* [Pfenning 2004]. Preservation dictates that if a variable *x* has type *t*, and *x* evaluates to a value *v*, then *v* also has type *t*. Progress tells us that evaluation of an expression does not get stuck in any unexpected way: either we have a value (and are done), or there is a way to proceed. In general, type safety implies that any operation on a particular type results in another value of that type. C was derived from two typeless languages and still shows many characteristics of a typeless or weakly typed language. For example, it is possible to use an explicit cast in C to convert from a pointer to one type to a pointer to a different type. If the resulting pointer is dereferenced, the results are undefined. Operations can legally act on signed and unsigned integers of differing lengths using implicit conversions and producing unrepresentable results. This lack of type safety leads to a wide range of security flaws and vulnerabilities.

For these reasons, the onus is on the C programmer to develop code that is free from undefined behaviors, with or without the help of the compiler.

In summary, C is a popular language that in many cases may be the language of choice for various applications, although it has characteristics that are commonly misused, resulting in security flaws. Some of these problems could be addressed as the language standard, compilers, and tools evolve. In the short term, the best hope for improvement is in educating developers in how to program securely by recognizing common security flaws and applying appropriate mitigations. In the long term, improvements must be made in the C language standard and implemented in compliant compilers and libraries for C to remain a viable language for developing secure systems.

Legacy Code

A significant amount of legacy C code was created (and passed on) before the standardization of the language. For example, Sun's external data representation (XDR) libraries are implemented almost completely in K&R C. Legacy C code is at higher risk for security flaws because of the looser compiler standards and is harder to secure because of the resulting coding style.

Other Languages

Because of these inherent problems with C, many security professionals recommend using other languages, such as Java. Although Java addresses many of the problems with C, it is still susceptible to implementation-level, as well as design-level, security flaws. Java's ability to operate with applications and libraries written in other languages using the Java Native Interface (JNI) allows systems to be composed using both Java and C or C++ components.

Adopting Java is often not a viable option because of an existing investment in C source code, programming expertise, and development environments. C may also be selected for performance or other reasons not pertaining to security. For whatever reason, when programs are developed in C and C++, the burden of producing secure code is placed largely on the programmer.

Another alternative to using C is to use a C dialect, such as Cyclone [Grossman 2005]. Cyclone was designed to provide the safety guarantee of Java (no valid program can commit a safety violation) while keeping C's syntax, types, semantics, and idioms intact. Cyclone is currently supported on 32-bit Intel architecture (IA-32) Linux and on Windows using Cygwin.[12]

Despite these characteristics, Cyclone may not be an appropriate choice for industrial applications because of the relative unpopularity of the language and consequent lack of tooling and programmers.

D is a general-purpose systems and applications programming language. D is based largely on the C++ language but drops features such as C source code compatibility and link compatibility with C++, allowing D to provide syntactic and semantic constructs that eliminate or at least reduce common programming mistakes [Alexandrescu 2010].

1.4 Development Platforms

Software vulnerabilities can be viewed at varying levels of abstraction. At higher levels of abstraction, software vulnerabilities can be common to multiple languages and multiple operating system environments. This book focuses on software flaws that are easily introduced in general C and C++ programming. Vulnerabilities often involve interactions with the environment and so are difficult to describe without assuming a particular operating system. Differences in compilation, linkage, and execution can lead to significantly different exploits and significantly different mitigation strategies.

12. Cygwin is a Linux-like environment for Windows.

To better illustrate vulnerabilities, exploits, and mitigations, this book focuses on the Microsoft Windows and Linux operating systems. These two operating systems were selected because of their popularity, broad adoption in critical infrastructure, and proclivity for vulnerabilities. The vulnerability of operating system software has been quantitatively assessed by O. H. Alhazmi and Y. K. Malaiya from Colorado State University [Alhazmi 2005a].

Operating Systems

Microsoft Windows. Many of the examples in this book are based on the Microsoft Windows family of operating system products, including Windows 7, Windows Vista, Windows XP, Windows Server 2003, Windows 2000, Windows Me, Windows 98, Windows 95, Windows NT Workstation, Windows NT Server, and other products.

Linux. Linux is a free UNIX derivative created by Linus Torvalds with the assistance of developers around the world. Linux is available for many different platforms but is commonly used on Intel-based machines.

Compilers

The choice of compiler and associated runtime has a large influence on the security of a program. The examples in this book are written primarily for Visual C++ on Windows and GCC on Linux, which are described in the following sections.

Visual C++. Microsoft's Visual C++ is the predominant C and C++ compiler on Windows platforms. Visual C++ is actually a family of compiler products that includes Visual Studio 2012, Visual Studio 2010, Visual Studio 2008, Visual Studio 2005, and older versions. These versions are all in widespread use and vary in functionality, including the security features provided. In general, the newer versions of the compiler provide more, and more advanced, security features. Visual Studio 2012, for example, includes improved support for the C++11 standard.

GCC. The GNU Compiler Collection, or GCC, includes front ends for C, C++, Objective-C, Fortran, Java, and Ada, as well as libraries for these languages. The GCC compilers are the predominant C and C++ compilers for Linux platforms.

GCC supports three versions of the C Standard: C89, AMD1, and C99. By default, the GCC compiler adheres to the ANSI (ISO C89) standard plus

GNU extensions. The GCC compiler also supports an `-std` flag that allows the user to specify the language standard when compiling C programs. Currently, the GCC compiler does not fully support the ISO C99 specification, with several features being listed as missing or broken.[13] GCC also provides limited, incomplete support for parts of the C11 standard.

1.5 Summary

It is no secret that common, everyday software defects cause the majority of software vulnerabilities. This is particularly true of C and C++, as the design of these languages assumes a level of expertise from developers that is not always present. The results are numerous delivered defects, some of which can lead to vulnerabilities. The software developers then respond to vulnerabilities found by users (some with malicious intent), and cycles of patch and install follow. However, patches are so numerous that system administrators cannot keep up with their installation. Often the patches themselves contain security defects. The strategy of responding to security defects is not working. A strategy of prevention and early security defect removal is needed.

Even though the principal causes of security issues in software are defects in the software, defective software is commonplace. The most widely used operating systems have from one to two defects per thousand lines of code, contain several million lines of code, and therefore typically have thousands of defects [Davis 2003]. Application software, while not as large, has a similar number of defects per thousand lines of code. While not every defect is a security concern, if only 1 or 2 percent lead to security vulnerabilities, the risk is substantial.

Alan Paller, director of research at the SANS Institute, expressed frustration that "everything on the [SANS Institute Top 20 Internet Security] vulnerability list is a result of poor coding, testing and sloppy software engineering. These are not 'bleeding edge' problems, as an innocent bystander might easily assume. Technical solutions for all of them exist, but they are simply not implemented" [Kirwan 2004].

Understanding the sources of vulnerabilities and learning to program securely are essential to protecting the Internet and ourselves from attack. Reducing security defects requires a disciplined engineering approach based on sound design principles and effective quality management practices.

13. See http://gcc.gnu.org/c99status.html for more information.

1.6 Further Reading

AusCERT surveys threats across a broad cross section of Australian industry, including public- and private-sector organizations [AusCERT 2006]. Bill Fithen and colleagues provide a more formal model for software vulnerabilities [Fithen 2004]. The Insider Threat Study report [Randazzo 2004], conducted by the U.S. Secret Service and the CERT/CC, provides a comprehensive analysis of insider actions by analyzing both the behavioral and technical aspects of the threats.

Bruce Schneier goes much further in his book *Secrets and Lies* [Schneier 2004] in explaining the context for the sometimes narrowly scoped software security topics detailed in this book.

Operational security is not covered in detail in this book but is the subject of *The CERT Guide to System and Network Security Practices* [Allen 2001]. The intersection of software development and operational security is best covered by Mark G. Graff and Kenneth R. van Wyk in *Secure Coding: Principles & Practices* [Graff 2003].

Chapter 2

Strings

with Dan Plakosh, Jason Rafail, and Martin Sebor[1]

But evil things, in robes of sorrow,
Assailed the monarch's high estate.

—Edgar Allan Poe,
"The Fall of the House of Usher"

2.1 Character Strings

Strings from sources such as command-line arguments, environment variables, console input, text files, and network connections are of special concern in secure programming because they provide means for external input to influence the behavior and output of a program. Graphics- and Web-based applications, for example, make extensive use of text input fields, and because of standards like XML, data exchanged between programs is increasingly in string form as well. As a result, weaknesses in string representation, string management, and string manipulation have led to a broad range of software vulnerabilities and exploits.

1. Daniel Plakosh is a senior member of the technical staff in the CERT Program of Carnegie Mellon's Software Engineering Institute (SEI). Jason Rafail is a Senior Cyber Security Consultant at Impact Consulting Solutions. Martin Sebor is a Technical Leader at Cisco Systems.

Strings are a fundamental concept in software engineering, but they are not a built-in type in C or C++. The standard C library supports strings of type `char` and wide strings of type `wchar_t`.

String Data Type

A string consists of a contiguous sequence of characters terminated by and including the first null character. A pointer to a string points to its initial character. The length of a string is the number of bytes preceding the null character, and the value of a string is the sequence of the values of the contained characters, in order. Figure 2.1 shows a string representation of "hello."

Strings are implemented as arrays of characters and are susceptible to the same problems as arrays.

As a result, secure coding practices for arrays should also be applied to null-terminated character strings; see the "Arrays (ARR)" chapter of *The CERT C Secure Coding Standard* [Seacord 2008]. When dealing with character arrays, it is useful to define some terms:

Bound
The number of elements in the array.

Lo
The address of the first element of the array.

Hi
The address of the last element of the array.

TooFar
The address of the one-too-far element of the array, the element just past the Hi element.

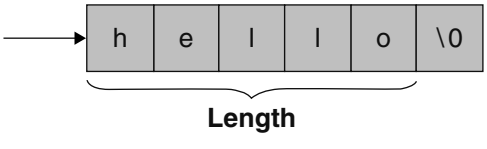

Figure 2.1 String representation of "hello"

Target size (Tsize)
Same as `sizeof(array)`.

The C Standard allows for the creation of pointers that point one past the last element of the array object, although these pointers cannot be dereferenced without invoking undefined behavior. When dealing with strings, some extra terms are also useful:

Null-terminated
At or before Hi, the null terminator is present.

Length
Number of characters prior to the null terminator.

Array Size. One of the problems with arrays is determining the number of elements. In the following example, the function `clear()` uses the idiom `sizeof(array) / sizeof(array[0])` to determine the number of elements in the array. However, `array` is a pointer type because it is a parameter. As a result, `sizeof(array)` is equal to `sizeof(int *)`. For example, on an architecture (such as x86-32) where `sizeof(int) == 4` and `sizeof(int *) == 4`, the expression `sizeof(array) / sizeof(array[0])` evaluates to 1, regardless of the length of the array passed, leaving the rest of the array unaffected.

```
01  void clear(int array[]) {
02    for (size_t i = 0; i < sizeof(array) / sizeof(array[0]); ++i) {
03       array[i] = 0;
04     }
05  }
06
07  void dowork(void) {
08    int dis[12];
09
10    clear(dis);
11    /* ... */
12  }
```

This is because the `sizeof` operator yields the size of the adjusted (pointer) type when applied to a parameter declared to have array or function type. The `strlen()` function can be used to determine the length of a properly null-terminated character string but not the space available in an array. *The CERT*

C Secure Coding Standard [Seacord 2008] includes "ARR01-C. Do not apply the `sizeof` operator to a pointer when taking the size of an array," which warns against this problem.

The characters in a string belong to the character set interpreted in the execution environment—the *execution character set*. These characters consist of a *basic character set*, defined by the C Standard, and a set of zero or more *extended characters*, which are not members of the basic character set. The values of the members of the execution character set are implementation defined but may, for example, be the values of the 7-bit U.S. ASCII character set.

C uses the concept of a *locale*, which can be changed by the `setlocale()` function, to keep track of various conventions such as language and punctuation supported by the implementation. The current locale determines which characters are available as extended characters.

The basic execution character set includes the 26 *uppercase* and 26 *lowercase* letters of the Latin alphabet, the 10 decimal digits, 29 graphic characters, the space character, and control characters representing horizontal tab, vertical tab, form feed, alert, backspace, carriage return, and newline. The representation of each member of the basic character set fits in a single byte. A byte with all bits set to 0, called the *null character*, must exist in the basic execution character set; it is used to terminate a character string.

The execution character set may contain a large number of characters and therefore require multiple bytes to represent some individual characters in the extended character set. This is called a *multibyte* character set. In this case, the basic characters must still be present, and each character of the basic character set is encoded as a single byte. The presence, meaning, and representation of any additional characters are locale specific. A string may sometimes be called a *multibyte string* to emphasize that it might hold multibyte characters. These are not the same as wide strings in which each character has the same length.

A multibyte character set may have a *state-dependent encoding*, wherein each sequence of multibyte characters begins in an *initial shift state* and enters other *locale-specific shift states* when specific multibyte characters are encountered in the sequence. While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state. The interpretation for subsequent bytes in the sequence is a function of the current shift state.

UTF-8

UTF-8 is a multibyte character set that can represent every character in the Unicode character set but is also backward compatible with the 7-bit U.S. ASCII character set. Each UTF-8 character is represented by 1 to 4 bytes (see Table 2.1). If the character is encoded by just 1 byte, the high-order bit is 0 and the other bits give the code value (in the range 0 to 127). If the character