SEI SERIES • A CERT® BOOK

Secure Coding in C and C++

SECOND EDITION

Robert C. Seacord

Foreword by Richard D. Pethia
CERT Director

Secure Coding in C and C++

Second Edition

Secure Coding in C and C++

Second Edition

Robert C. Seacord

Addison-Wesley

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

Software Engineering Institute | Carnegie Mellon

The SEI Series in Software Engineering

Library of Congress Cataloging Control Number: 2013932290

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

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

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

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

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

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

With the large number of vulnerabilities being discovered each year, administrators are increasingly overwhelmed with patching existing systems. Patches can be difficult to apply and might have unexpected side effects. After a vendor releases a security patch it can take months, or even years, before 90 to 95 percent of the vulnerable computers are fixed.

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

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

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

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

About Secure Coding in C and C++

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

I have worked with Robert since he first joined the SEI in April, 1987. Robert is a skilled and knowledgeable software engineer who has proven himself adept at detailed software vulnerability analysis and in communicating his observations and discoveries. As a result, this book provides a meticulous treatment of the most common problems faced by software developers and provides practical solutions. Robert's extensive background in software development has also made him sensitive to trade-offs in performance, usability, and other quality attributes that must be balanced when developing secure code. In addition to Robert's abilities, this book also represents the knowledge collected and distilled by CERT operations and the exceptional work of the CERT/CC vulnerability analysis team, the CERT operations staff, and the editorial and support staff of the Software Engineering Institute.

—Richard D. Pethia
CERT Director

This page intentionally left blank

Preface

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

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

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

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

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

Audience

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

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

Organization and Content

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

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

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

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

This book contains the following chapters:

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

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

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

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

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

Acknowledgments

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

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

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

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

I would also like to thank the following reviewers for their thoughtful comments and insights: Tad Anderson, John Benito, William Bulley, Corey Cohen, Will Dormann, William Fithen, Robin Eric Fredericksen, Michael Howard, Michael Kaelbling, Amit Kalani, John Lambert, Jeffrey Lanza, David LeBlanc, Ken MacInnis, Gary McGraw, Randy Meyers, Philip Miller, Patrick Mueller, Dave Mundie, Craig Partridge, Brad Rubbo, Tim Shimeall, Michael Wang, and Katie Washok.

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

About the Author

Robert C. Seacord is the Secure Coding Technical Manager in the CERT Program of Carnegie Mellon's Software Engineering Institute (SEI) in Pittsburgh, Pennsylvania. The CERT Program is a trusted provider of operationally relevant cybersecurity research and innovative and timely responses to our nation's cybersecurity challenges. The Secure Coding Initiative works with software developers and software development organizations to eliminate vulnerabilities resulting from coding errors before they are deployed. Robert is also an adjunct professor in the School of Computer Science and the Information Networking Institute at Carnegie Mellon University. He is the author of *The CERT C Secure Coding Standard* (Addison-Wesley, 2008) and coauthor of *Building Systems from Commercial Components* (Addison-Wesley, 2002), *Modernizing Legacy Systems* (Addison-Wesley, 2003), and *The CERT Oracle Secure Coding Standard for Java* (Addison-Wesley, 2011). He has also published more than forty papers on software security, component-based software engineering, Web-based system design, legacy-system modernization, component repositories and search engines, and user interface design and development. Robert has been teaching Secure Coding in C and C++ to private industry, academia, and government since 2005. He started programming professionally for IBM in 1982, working in communications and operating system software, processor development, and software engineering. Robert has also worked at the X Consortium, where he developed and maintained code for the Common Desktop Environment and the X Window System. He represents Carnegie Mellon University (CMU) at the ISO/IEC JTC1/SC22/WG14 international standardization working group for the C programming language.

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

Chapter 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.

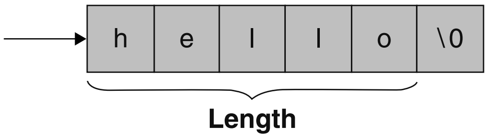


Figure 2.1 String representation of "hello"

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

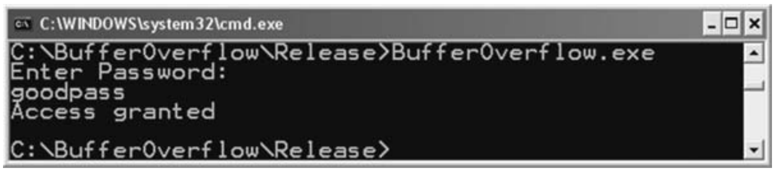
Example 2.3 is a simple program that checks a user password (which should be considered tainted data) and grants or denies access.

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);
  }
}
```

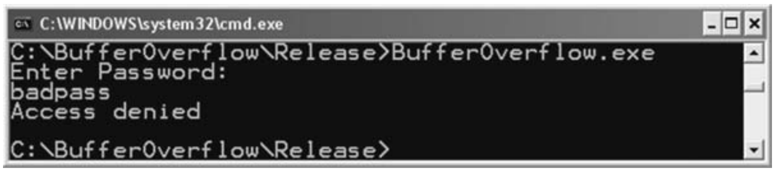
This program shows how strings can be misused and is not an exemplar for password checking. The `IsPasswordOK` program starts in the `main()` function. The first line executed is the `puts()` call that prints out a string literal. The `puts()` function, defined in the C Standard as a character output 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.

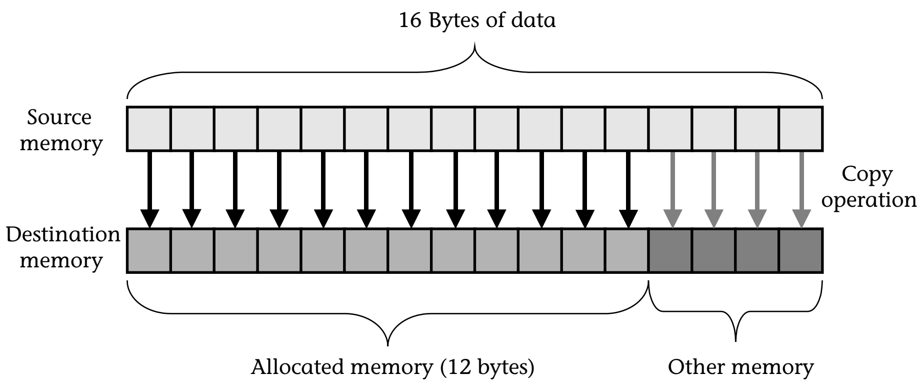


Figure 2.4 Copying 16 bytes of data into a 12-byte array

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.

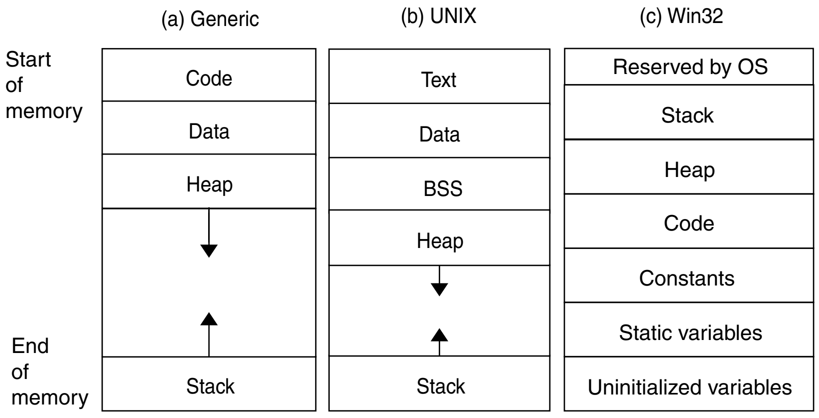
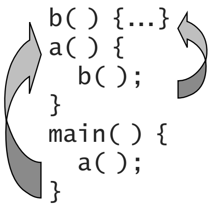


Figure 2.5 Process memory organization

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

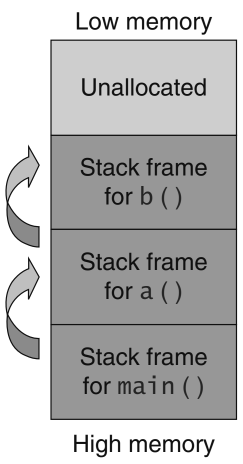


Figure 2.7 Calling a subroutine

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

```
01  void foo(int, char *); // function prototype
02
```

```
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.

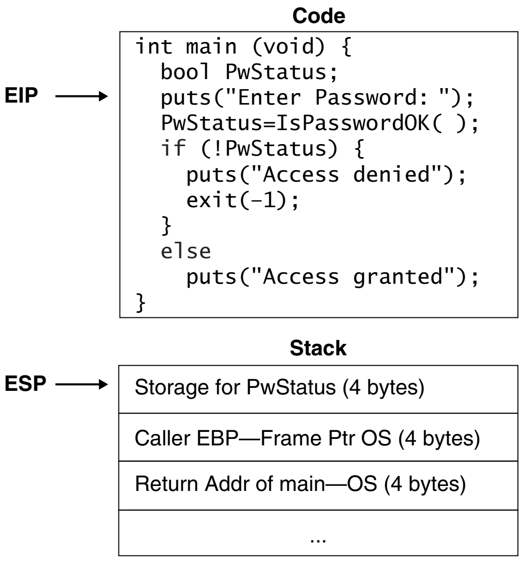
The operating system (OS) or a standard start-up sequence puts the return address from `main()` on the stack. On entry, `main()` saves the old incoming frame pointer, which again comes from the operating system or a standard start-up sequence. Before the call to the `IsPasswordOK()` function, the stack 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()`.

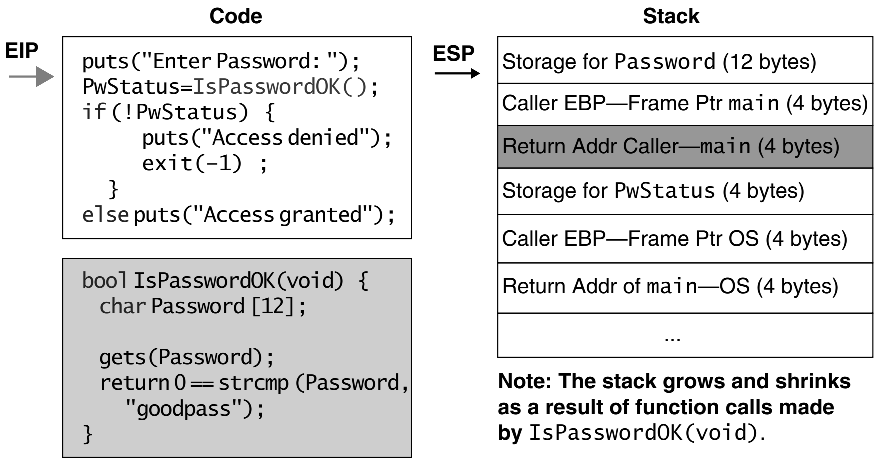
After the program returns from the `IsPasswordOK()` function, the stack is restored to its initial state, as in Figure 2.10.

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


Code
EIP
ESP
Stack
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");
}
```

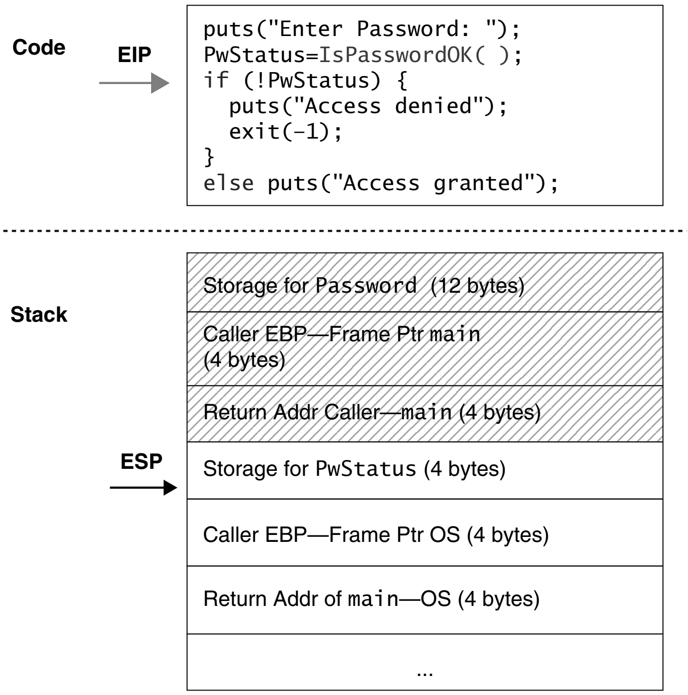
Figure 2.8 The stack before `IsPasswordOK()` is called


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

```
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");
}
```

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

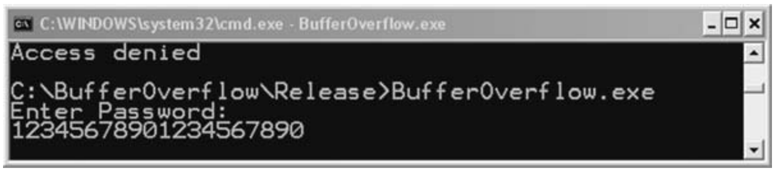

Code
EIP
Stack
ESP
Storage for Password (12 bytes)
Caller EBP—Frame Ptr main (4 bytes)
Return Addr Caller—main (4 bytes)
Storage for PwStatus (4 bytes)
Caller EBP—Frame Ptr OS (4 bytes)
Return Addr of main—OS (4 bytes)
...

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

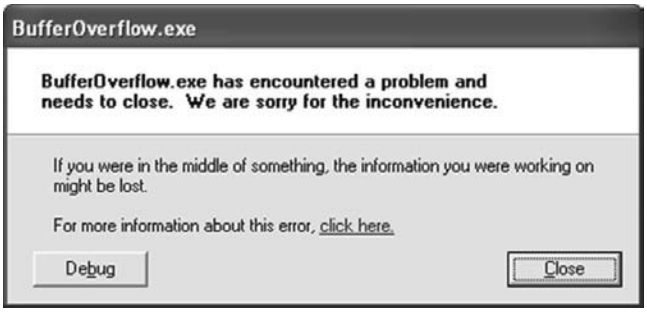
Figure 2.10 Stack restored to initial state

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
```



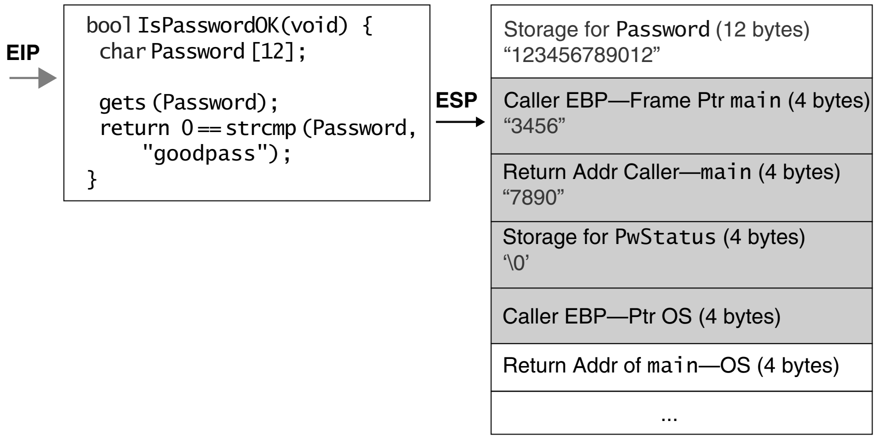
BufferOverflow.exe

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.



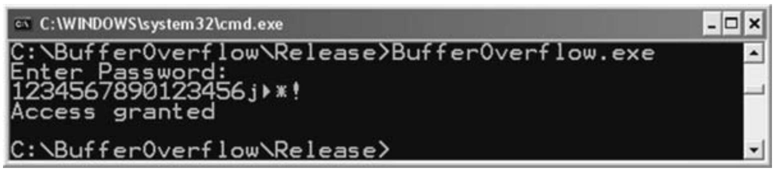
```
bool IsPasswordOK(void) {
  char Password[12];

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

EIP
ESP
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>
```

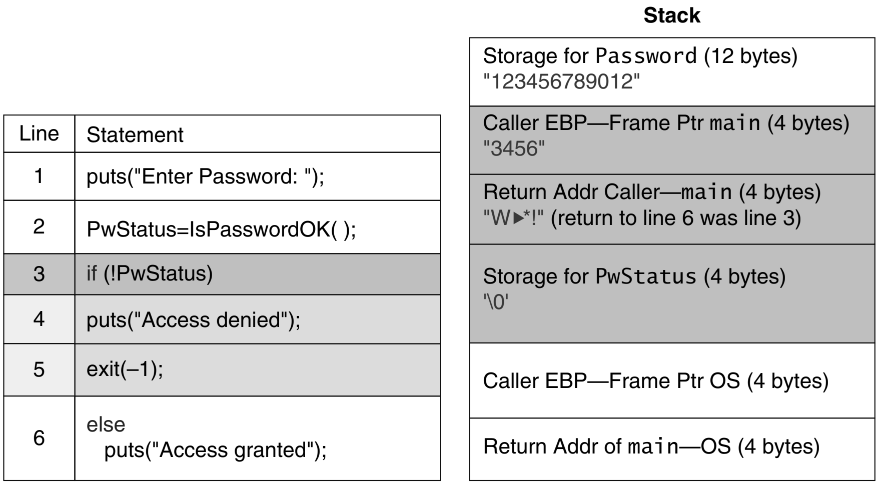
Figure 2.13 Unexpected results from a carefully crafted input string

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 */
```

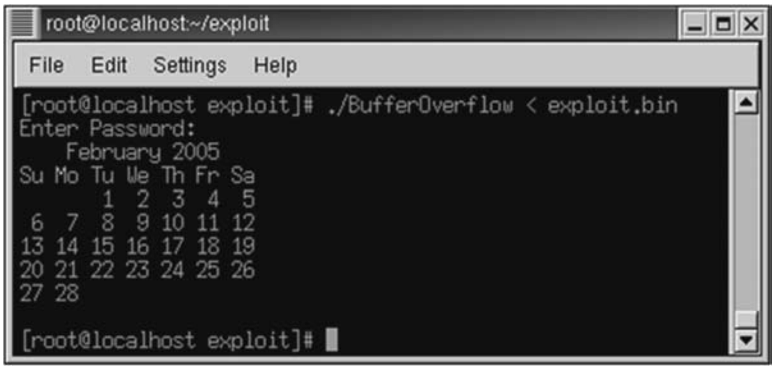
The `lea` instruction used in this example stands for "load effective address." The `lea` instruction computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with 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

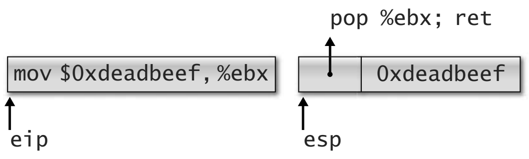


Figure 2.16 Gadget built with return-oriented programming

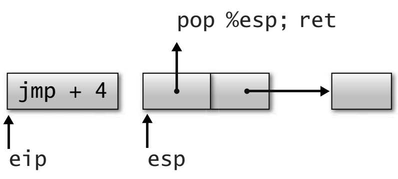


Figure 2.17 Unconditional branching in x86-32 assembly language (left) and return-oriented programming idioms

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.

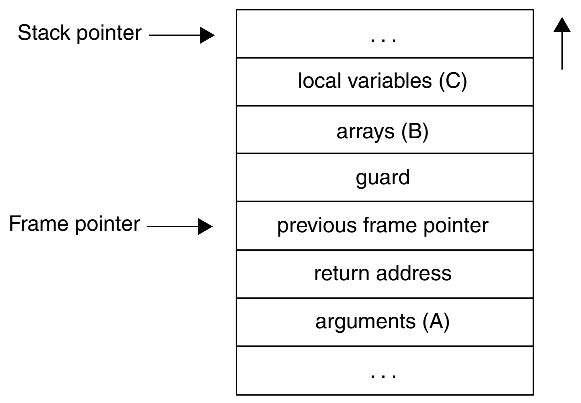


Figure 2.18 Stack-Smashing Protector (SSP) stack structure

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;
}
```

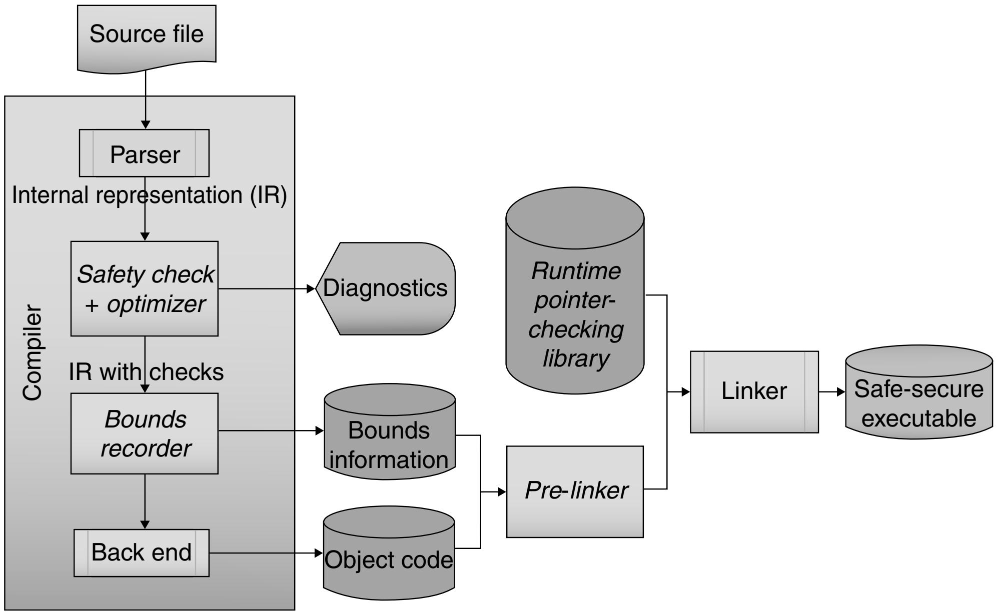


Figure 2.19 A possible Safe-Secure C/C++ (SSCC) implementation

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.

Index

Note: Page numbers followed by *f* and *t* indicate figures and tables, respectively. Footnotes are indicated by *n*.