Secure Coding in C and C++

Second Edition

Robert C. Seacord

Foreword by Richard D. Pethia
CERT Director

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

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

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

Foreword xvii
Preface xxi
Acknowledgments xxv
About the Author xxvii

Chapter 1 Running with Scissors 1
1.1 Gauging the Threat 5
What Is the Cost? 6
Who Is the Threat? 8
Software Security 11
1.2 Security Concepts 12
Security Policy 14
Security Flaws 14
Vulnerabilities 15
Exploits 16
Mitigations 17
1.3 C and C++ 17
A Brief History 19
What Is the Problem with C? 21
Legacy Code 24
Other Languages 25
1.4 Development Platforms 25
Operating Systems 26
Compilers 26
1.5 Summary 27
1.6 Further Reading 28

Chapter 2  Strings 29

2.1 Character Strings 29
  String Data Type 30
  UTF-8 32
  Wide Strings 33
  String Literals 34
  Strings in C++ 36
  Character Types 37
  Sizing Strings 39

2.2 Common String Manipulation Errors 42
  Improperly Bounded String Copies 42
  Off-by-One Errors 47
  Null-Termination Errors 48
  String Truncation 49
  String Errors without Functions 49

2.3 String Vulnerabilities and Exploits 50
  Tainted Data 51
  Security Flaw: IsPasswordOK 52
  Buffer Overflows 53
  Process Memory Organization 54
  Stack Management 55
  Stack Smashing 59
  Code Injection 64
  Arc Injection 69
  Return-Oriented Programming 71

2.4 Mitigation Strategies for Strings 72
  String Handling 73
  C11 Annex K Bounds-Checking Interfaces 73
  Dynamic Allocation Functions 76
  C++ std::basic_string 80
  Invalidating String Object References 81
  Other Common Mistakes in basic_string Usage 83

2.5 String-Handling Functions 84
  gets() 84
  C99 84
  C11 Annex K Bounds-Checking Interfaces: gets_s() 86
  Dynamic Allocation Functions 87
  strcpy() and strcat() 89
  C99 89
  strncpy() and strncat() 93
  memcpy() and memmove() 100
  strlen() 100
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>Runtime Protection Strategies</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>Detection and Recovery</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>Input Validation</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>Object Size Checking</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>Visual Studio Compiler-Generated Runtime Checks</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>Stack Canaries</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>Stack-Smashing Protector (ProPolice)</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>Operating System Strategies</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>Detection and Recovery</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>Nonexecutable Stacks</td>
<td>113</td>
</tr>
<tr>
<td></td>
<td>W^X</td>
<td>113</td>
</tr>
<tr>
<td></td>
<td>PaX</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>Future Directions</td>
<td>116</td>
</tr>
<tr>
<td>2.7</td>
<td>Notable Vulnerabilities</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>Remote Login</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>Kerberos</td>
<td>118</td>
</tr>
<tr>
<td>2.8</td>
<td>Summary</td>
<td>118</td>
</tr>
<tr>
<td>2.9</td>
<td>Further Reading</td>
<td>120</td>
</tr>
</tbody>
</table>

**Chapter 3**  
**Pointer Subterfuge**  
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Data Locations</td>
<td>122</td>
</tr>
<tr>
<td>3.2</td>
<td>Function Pointers</td>
<td>123</td>
</tr>
<tr>
<td>3.3</td>
<td>Object Pointers</td>
<td>124</td>
</tr>
<tr>
<td>3.4</td>
<td>Modifying the Instruction Pointer</td>
<td>125</td>
</tr>
<tr>
<td>3.5</td>
<td>Global Offset Table</td>
<td>127</td>
</tr>
<tr>
<td>3.6</td>
<td>The .dtors Section</td>
<td>129</td>
</tr>
<tr>
<td>3.7</td>
<td>Virtual Pointers</td>
<td>131</td>
</tr>
<tr>
<td>3.8</td>
<td>The atexit() and on_exit() Functions</td>
<td>133</td>
</tr>
<tr>
<td>3.9</td>
<td>The longjmp() Function</td>
<td>134</td>
</tr>
<tr>
<td>3.10</td>
<td>Exception Handling</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td>Structured Exception Handling</td>
<td>137</td>
</tr>
<tr>
<td></td>
<td>System Default Exception Handling</td>
<td>139</td>
</tr>
<tr>
<td>3.11</td>
<td>Mitigation Strategies</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>Stack Canaries</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>W^X</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>Encoding and Decoding Function Pointers</td>
<td>140</td>
</tr>
<tr>
<td>3.12</td>
<td>Summary</td>
<td>142</td>
</tr>
<tr>
<td>3.13</td>
<td>Further Reading</td>
<td>143</td>
</tr>
</tbody>
</table>

**Chapter 4**  
**Dynamic Memory Management**  
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>C Memory Management</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>C Standard Memory Management Functions</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>Alignment</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>alloca() and Variable-Length Arrays</td>
<td>149</td>
</tr>
</tbody>
</table>
5.3 Integer Conversions 246
Converting Integers 246
Integer Conversion Rank 246
Integer Promotions 247
Usual Arithmetic Conversions 249
Conversions from Unsigned Integer Types 250
Conversions from Signed Integer Types 253
Conversion Implications 256

5.4 Integer Operations 256
Assignment 258
Addition 260
Subtraction 267
Multiplication 269
Division and Remainder 274
Shifts 279

5.5 Integer Vulnerabilities 283
Vulnerabilities 283
Wraparound 283
Conversion and Truncation Errors 285
Nonexceptional Integer Logic Errors 287

5.6 Mitigation Strategies 288
Integer Type Selection 289
Abstract Data Types 291
Arbitrary-Precision Arithmetic 292
Range Checking 293
Precondition and Postcondition Testing 295
Secure Integer Libraries 297
Overflow Detection 299
Compiler-Generated Runtime Checks 300
Verifiably In-Range Operations 301
As-If Infinitely Ranged Integer Model 303
Testing and Analysis 304

5.7 Summary 307

Chapter 6 Formatted Output 309
6.1 Variadic Functions 310
6.2 Formatted Output Functions 313
Format Strings 314
## Contents

- Corrupted Values  
- Volatile Objects  

### 7.5 Mitigation Strategies  
- Memory Model  
- Synchronization Primitives  
- Thread Role Analysis (Research)  
- Immutable Data Structures  
- Concurrent Code Properties  

### 7.6 Mitigation Pitfalls  
- Deadlock  
- Prematurely Releasing a Lock  
- Contention  
- The ABA Problem  

### 7.7 Notable Vulnerabilities  
- DoS Attacks in Multicore Dynamic Random-Access Memory (DRAM) Systems  
- Concurrency Vulnerabilities in System Call Wrappers  

### 7.8 Summary  

---

## Chapter 8 File I/O  

### 8.1 File I/O Basics  
- File Systems  
- Special Files  

### 8.2 File I/O Interfaces  
- Data Streams  
- Opening and Closing Files  
- POSIX  
- File I/O in C++  

### 8.3 Access Control  
- UNIX File Permissions  
- Process Privileges  
- Changing Privileges  
- Managing Privileges  
- Managing Permissions  

### 8.4 File Identification  
- Directory Traversal  
- Equivalence Errors  
- Symbolic Links  
- Canonicalization  
- Hard Links  
- Device Files  
- File Attributes  

### 8.5 Race Conditions  
- Time of Check, Time of Use (TOCTOU)
### Chapter 9  **Recommended Practices**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>The Security Development Lifecycle</td>
<td>474</td>
</tr>
<tr>
<td></td>
<td>TSP-Secure</td>
<td>477</td>
</tr>
<tr>
<td></td>
<td>Planning and Tracking</td>
<td>477</td>
</tr>
<tr>
<td></td>
<td>Quality Management</td>
<td>479</td>
</tr>
<tr>
<td>9.2</td>
<td>Security Training</td>
<td>480</td>
</tr>
<tr>
<td>9.3</td>
<td>Requirements</td>
<td>481</td>
</tr>
<tr>
<td></td>
<td>Secure Coding Standards</td>
<td>481</td>
</tr>
<tr>
<td></td>
<td>Security Quality Requirements Engineering</td>
<td>483</td>
</tr>
<tr>
<td></td>
<td>Use/Misuse Cases</td>
<td>485</td>
</tr>
<tr>
<td>9.4</td>
<td>Design</td>
<td>486</td>
</tr>
<tr>
<td></td>
<td>Secure Software Development Principles</td>
<td>488</td>
</tr>
<tr>
<td></td>
<td>Threat Modeling</td>
<td>493</td>
</tr>
<tr>
<td></td>
<td>Analyze Attack Surface</td>
<td>494</td>
</tr>
<tr>
<td></td>
<td>Vulnerabilities in Existing Code</td>
<td>495</td>
</tr>
<tr>
<td></td>
<td>Secure Wrappers</td>
<td>496</td>
</tr>
<tr>
<td></td>
<td>Input Validation</td>
<td>497</td>
</tr>
<tr>
<td></td>
<td>Trust Boundaries</td>
<td>498</td>
</tr>
<tr>
<td></td>
<td>Blacklisting</td>
<td>501</td>
</tr>
<tr>
<td></td>
<td>Whitelisting</td>
<td>502</td>
</tr>
<tr>
<td></td>
<td>Testing</td>
<td>503</td>
</tr>
<tr>
<td>9.5</td>
<td>Implementation</td>
<td>503</td>
</tr>
<tr>
<td></td>
<td>Compiler Security Features</td>
<td>503</td>
</tr>
<tr>
<td></td>
<td>As-If Infinitely Ranged (AIR) Integer Model</td>
<td>505</td>
</tr>
<tr>
<td></td>
<td>Safe-Secure C/C++</td>
<td>505</td>
</tr>
<tr>
<td></td>
<td>Static Analysis</td>
<td>506</td>
</tr>
<tr>
<td></td>
<td>Source Code Analysis Laboratory (SCALe)</td>
<td>510</td>
</tr>
<tr>
<td></td>
<td>Defense in Depth</td>
<td>511</td>
</tr>
<tr>
<td>9.6</td>
<td>Verification</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>Static Analysis</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>Penetration Testing</td>
<td>513</td>
</tr>
<tr>
<td></td>
<td>Fuzz Testing</td>
<td>513</td>
</tr>
<tr>
<td></td>
<td>Code Audits</td>
<td>515</td>
</tr>
<tr>
<td></td>
<td>Developer Guidelines and Checklists</td>
<td>516</td>
</tr>
<tr>
<td>Contents</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>Independent Security Review</td>
<td>516</td>
<td></td>
</tr>
<tr>
<td>Attack Surface Review</td>
<td>517</td>
<td></td>
</tr>
<tr>
<td>9.7 Summary</td>
<td>518</td>
<td></td>
</tr>
<tr>
<td>9.8 Further Reading</td>
<td>518</td>
<td></td>
</tr>
<tr>
<td><strong>References</strong></td>
<td>519</td>
<td></td>
</tr>
<tr>
<td><strong>Acronyms</strong></td>
<td>539</td>
<td></td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td>545</td>
<td></td>
</tr>
</tbody>
</table>
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
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](http://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.
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,
Acknowledgments


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

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:

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bound</td>
<td>The number of elements in the array.</td>
</tr>
<tr>
<td>Lo</td>
<td>The address of the first element of the array.</td>
</tr>
<tr>
<td>Hi</td>
<td>The address of the last element of the array.</td>
</tr>
<tr>
<td>TooFar</td>
<td>The address of the one-too-far element of the array, the element just past the Hi element.</td>
</tr>
</tbody>
</table>

![Figure 2.1 String representation of “hello”](hello_string_representation)
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.

```c
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
is encoded by a sequence of more than 1 byte, the first byte has as many lead-
ing 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 remain-
ing 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-re-
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 rep-
resent a wide character. The problem of sizing wide strings is covered in the
section “Sizing Strings.”

---

### Table 2.1 Well-Formed UTF-8 Byte Sequences

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

Source: [Unicode 2012]
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:

```c
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:

```c
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.
2.1 Character Strings

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 *`. 
Strings

1. size_t len;
2. char cstr[] = "char string";
3. signed char scstr[] = "signed char string";
4. unsigned char ucstr[] = "unsigned char string";
5. len = strlen(cstr);
6. len = strlen(scstr);  /* warns when char is unsigned */
7. 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 \texttt{sizeof 'a'} is not equal to \texttt{sizeof x} when \texttt{x} is a variable of type \texttt{char}.

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

\textbf{unsigned char}

The \texttt{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 \texttt{fwrite()}. Unlike other integer types, \texttt{unsigned char} has the unique property that values stored in objects of type \texttt{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 \texttt{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 \texttt{unsigned char} (for example, via \texttt{memcpy()}) and have their representation examined 1 byte at a time.

\textbf{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.

\section*{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 \textit{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:
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:

```c
1  wchar_t wide_str1[] = L"0123456789";
2  wchar_t *wide_str2 = (wchar_t *)malloc(strlen(wide_str1) + 1);
3  if (wide_str2 == NULL) {
4    /* handle error */
5  }
6  /* ... */
7  free(wide_str2);
8  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):

```
1  wchar_t wide_str1[] = L"0123456789";
2  wchar_t *wide_str3 = (wchar_t *)malloc(wcslen(wide_str1) + 1);
3  if (wide_str3 == NULL) {
4    /* handle error */
5  }
6  /* ... */
7  free(wide_str3);
8  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):

```
01  wchar_t wide_str1[] = L"0123456789";
02  wchar_t *wide_str2 = (wchar_t *)malloc((wcslen(wide_str1) + 1) * sizeof(wchar_t));
03  if (wide_str2 == NULL) {
04    /* handle error */
05  }
06  /* ... */
07  free(wide_str2);
08  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()

```c
#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:
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:

```
1 int main(int argc, char *argv[]) {
2   /* ...*/
3 } 
```

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:

```c
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.

```c
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.
2.2 Common String Manipulation Errors

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-1`st 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:

```cpp
1 #include <iostream>
2
3 int main(void) {
```
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::istream` 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

```cpp
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()`.
2.2 Common String Manipulation Errors

Example 2.2  Field width Member

```cpp
#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?

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

int main(void) {
    char s1[] = "0123456789";
    char s2[] = "0123456789999";
    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:

```c
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.

```c
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()`:

```c
1  char ntbs[NTBS_SIZE];
2 3  strncpy(ntbs, source, sizeof(ntbs)-1);
4  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:

```c
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

```c
01  bool IsPasswordOK(void) {
02    char Password[12];
03    
04    gets(Password);
05    return 0 == strcmp(Password, "goodpass");
06  }
07  
08  int main(void) {
09    bool PwStatus;
10    
11    puts("Enter password:");
12    PwStatus = IsPasswordOK();
13    if (PwStatus == false) {
14      puts("Access denied");
15      exit(-1);
16    }
17  }
```

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
<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.
2.3 String Vulnerabilities and Exploits

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.
2.3 String Vulnerabilities and Exploits

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.

Figure 2.5 Process memory organization

<table>
<thead>
<tr>
<th>Start of memory</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>End of memory</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td></td>
</tr>
<tr>
<td>BSS</td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
</tr>
<tr>
<td>Reserved by OS</td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>Constants</td>
<td></td>
</tr>
<tr>
<td>Static variables</td>
<td></td>
</tr>
<tr>
<td>Uninitialized variables</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.6 Stack management

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

```c
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:

```c
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
    sub esp, 28    # current stack pointer.
    /* ... */
    push ebp       # Allocates space for local variables.
    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.
}
```

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:

```c
/* ... */
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012FF4C</td>
<td>?</td>
<td>Last local variable—integer: LocalInt</td>
<td>4</td>
</tr>
<tr>
<td>0x0012FF50</td>
<td>?</td>
<td>First local variable—string: LocalChar</td>
<td>24</td>
</tr>
<tr>
<td>0x0012FF68</td>
<td>0x12FF80</td>
<td>Calling frame of calling function: main()</td>
<td>4</td>
</tr>
<tr>
<td>0x0012FF6C</td>
<td>0x401040</td>
<td>Return address of calling function: main()</td>
<td>4</td>
</tr>
<tr>
<td>0x0012FF70</td>
<td>1</td>
<td>First argument: MyInt (int)</td>
<td>4</td>
</tr>
<tr>
<td>0x0012FF74</td>
<td>0x40703C</td>
<td>Second argument: pointer toMyString (char *)</td>
<td>4</td>
</tr>
</tbody>
</table>
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.
int main (void) {
  bool PwStatus;
  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 2.8 The stack before IsPasswordOK() is called

Figure 2.9 Information in stack while IsPasswordOK() is executed
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.
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
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
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:

```
01 /* buf[12] */
02 00 00 00 00
03 00 00 00 00
04 00 00 00 00
05
06 /* %ebp */
07 00 00 00 00
08
09 /* return address */
10 78 fd ff bf
11
12 /* "/usr/bin/cal" */
13 2f 75 73 72
14 2f 62 69 6e
15 2f 63 61 6c
16 00 00 00 00
17
18 /* null pointer */
19 74 fd ff bf
20
21 /* NULL */
22 00 00 00 00
23
24 /* exploit code */
25 b0 0b /* mov $0xb, %eax */
26 8d 1c 24 /* lea (%esp), %ebx */
27 8d 4c 24 f0 /* lea -0x10(%esp), %ecx */
28 8b 54 24 ec /* mov -0x14(%esp), %edx */
29 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:

```
01  char buffer[128];
02
03  _Bool IsPasswordOK(void) {
04    char Password[12];
05
06    fgets(buffer, sizeof buffer, stdin);
07    if (buffer[ strlen(buffer) - 1 ] == '\n')
08      buffer[ strlen(buffer) - 1 ] = 0;
09    strcpy(Password, buffer);
10    return 0 == strcmp(Password, "goodpass");
11  }
12
13  int main(void) {
14    _Bool PwStatus;
15
16    puts("Enter password:");
17    PwStatus = IsPasswordOK();
18    if (!PwStatus) {
19      puts("Access denied");
20      exit(-1);
21    }
22    else
23      puts("Access granted");
24    return 0;
25  }
```

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.
The malicious argument in this case is in the binary file \texttt{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 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 \texttt{strcpy()} function returns, the stack is overwritten as shown in Table 2.3.

### Table 2.3  Corrupted Stack for the Call to \texttt{strcpy()}

<table>
<thead>
<tr>
<th>Row</th>
<th>Address</th>
<th>Content</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0xbffff9c0 - 0xbffff9cf</td>
<td>1234567890123456</td>
<td>Storage for Password (16 bytes) and padding</td>
</tr>
<tr>
<td>2</td>
<td>0xbfffff9d0 - 0xbfffff9db</td>
<td>789012345678</td>
<td>Additional padding</td>
</tr>
<tr>
<td>3</td>
<td>0xbffff9dc</td>
<td>(0xbfffff9e0)</td>
<td>New return address</td>
</tr>
<tr>
<td>4</td>
<td>0xbffff9e0</td>
<td>xor %eax,%eax</td>
<td>Sets eax to 0</td>
</tr>
</tbody>
</table>

(continues)
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

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

<table>
<thead>
<tr>
<th>Row</th>
<th>Address</th>
<th>Content</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0xbfffff9e2</td>
<td>mov %eax,0xbfffff9ff</td>
<td>Terminates pointer array with null pointer</td>
</tr>
<tr>
<td>6</td>
<td>0xbfffff9e7</td>
<td>mov $0xb,%al</td>
<td>Sets the code for the <code>execve()</code> function call</td>
</tr>
<tr>
<td>7</td>
<td>0xbfffff9e9</td>
<td>mov $0xbfffffa03,%ebx</td>
<td>Sets ebx to point to the first argument to <code>execve()</code></td>
</tr>
<tr>
<td>8</td>
<td>0xbfffff9ee</td>
<td>mov $0xbfffff9fb,%ecx</td>
<td>Sets ecx to point to the second argument to <code>execve()</code></td>
</tr>
<tr>
<td>9</td>
<td>0xbfffff9f3</td>
<td>mov 0xbfffff9ff,%edx</td>
<td>Sets edx to point to the third argument to <code>execve()</code></td>
</tr>
<tr>
<td>10</td>
<td>0xbfffff9f9</td>
<td>int $80</td>
<td>Invokes <code>execve()</code> system call</td>
</tr>
<tr>
<td>11</td>
<td>0xbfffff9fb</td>
<td>0xbfffff9ff</td>
<td>Array of argument strings passed to the new program</td>
</tr>
<tr>
<td>12</td>
<td>0xbfffff9ff</td>
<td>&quot;1111&quot;</td>
<td>Changed to 0x00000000 to terminate the pointer array and also used as the third argument</td>
</tr>
<tr>
<td>13</td>
<td>0xbfffa03</td>
<td>&quot;/usr/bin/cal\0&quot;</td>
<td>Command to execute</td>
</tr>
</tbody>
</table>
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

\[
\text{pop} \ %\text{ebx}; \\
\text{ret}
\]

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 $0x\text{deadbeef}$ 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
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 pro-
gramming 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 imple-
mented by a conditional branch out of the loop.

Hovav Shacham’s “The Geometry of Innocent Flesh on the Bone” [Sha-
cham 2007] contains a more complete tutorial on return-oriented program-
ning. 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 over-
flows 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 stra-
egies. A common approach is to consistently apply a secure technique to
string handling (a prevention strategy) and back it up with one or more run-
time detection and recovery schemes.
2.4 Mitigation Strategies for Strings

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()

```c
#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)

```c
01  #define __STDC_WANT_LIB_EXT1__ 1
02  #include <stdio.h>
03  #include <stdlib.h>
04
05  void get_y_or_n(void) {
06    char response[8];
07    size_t len = sizeof(response);
08    puts("Continue? [y] n: ");
09    if ((gets_s(response, len) == NULL) || (response[0] == 'n')) {
10       exit(0);
11    }
12  }
13 }
14
15  int main(void) {
16    constraint_handler_t oconstraint =
17      set_constraint_handler_s(ignore_handler_s);
18    get_y_or_n();
19  }
```

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()

```c
#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:

```
1  FILE *fmemopen(
2    void * restrict buf, size_t size, const char * restrict mode
3  );
4  FILE *open_memstream(
5    char ** restrict bufp, size_t * restrict sizep
6  );
```

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

```c
#include <stdio.h>

int main(void) {
    char *buf;
    size_t size;
    FILE *stream;

    stream = open_memstream(&buf, &size);
    if (stream == NULL) { /* handle error */);
    fprintf(stream, "hello");
    fflush(stream);
    printf("buf = '%s', size = %zu\n", buf, size);
    fprintf(stream, ", world");
    fclose(stream);
    printf("buf = '%s', size = %zu\n", buf, size);
    free(buf);
    return 0;
}
```

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:

```cpp
1  string str1 = "hello, ";
2  string str2 = "world";
3  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:
2.4 Mitigation Strategies for Strings

```cpp
#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:

```cpp
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:

```cpp
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 \texttt{loc} is invalidated after
the first call to \texttt{insert()}, and every subsequent call to \texttt{insert()}
results in undefined behavior. This problem can be easily repaired if the programmer is
aware of the issue:

\begin{verbatim}
01 char input[];
02 string email;
03 string::iterator loc = email.begin();
04 // copy into string converting ";" to " 
05 for (size_t i=0; i < strlen(input); ++i) {
06   if (input[i] != ';') {
07     loc = email.insert(loc, input[i]);
08   } else loc = email.insert(loc, ' ');
09   ++loc;
10 }
\end{verbatim}

In this version of the program, the value of the iterator \texttt{loc} is properly
updated as a result of each insertion, eliminating the undefined behavior.
Most checked standard template library (STL) implementations detect com-
mon 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 \texttt{basic_string} class generally protects against buffer overflow, but
there are still situations in which programming errors can lead to buffer over-
flows. While C++ generally throws an exception of type \texttt{std::out_of_range}
when an operation references memory outside the bounds of the string, for
maximum efficiency, the subscript member \texttt{std::string::operator[]} (which
does not perform bounds checking) does not. For example, the following pro-
gram fragment can result in a write outside the bounds of the storage allo-
cated to the \texttt{bs} string object if \texttt{f()} \texttt{\geq} \texttt{bs.size()}:

\begin{verbatim}
1  string bs("01234567");
2  size_t i = f();
3  bs[i] = '\0';
\end{verbatim}

The \texttt{at()} method behaves in a similar fashion to the index \texttt{operator[]} but
throws an \texttt{out_of_range} exception if \texttt{pos} \texttt{\geq} \texttt{size()}:

\begin{verbatim}
1  string bs("01234567");
2  try {
3    size_t i = f();
4    bs.at(i) = '\0';
5  }
\end{verbatim}
2.4 Mitigation Strategies for Strings

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

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.

```cpp
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()

```
01  char buf[LINE_MAX];
02  int ch;
03  char *p;
04
05  if (fgets(buf, sizeof(buf), stdin)) {
06    /* fgets succeeds, scan for newline character */
07    p = strchr(buf, '\n');
```
Unlike `gets()` 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()

```c
char buf[BUFSIZ];
int ch;
int index = 0;
int chars_read = 0;
while (((ch = getchar()) != \n'))
```
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:

```c
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, \texttt{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 \texttt{RSIZE_MAX} or if the pointer to the destination character array is \texttt{NULL}. If an error condition occurs, no input is performed and the character array is not modified. Otherwise, the \texttt{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 \texttt{stdin} using the \texttt{gets_s()} function.

\textbf{Example 2.11}  \hspace{1em} Reading from \texttt{stdin} Using \texttt{gets_s()}

\begin{verbatim}
1  char buf[BUFSIZ];
2 3  if (gets_s(buf, sizeof(buf)) == NULL) {
4    /* handle error */
5  }
\end{verbatim}

The \texttt{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 \texttt{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 \texttt{NULL}, sets the buffer to the null string, and clears the input stream to the next newline character.

The \texttt{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 \texttt{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 \texttt{fgets()} pay attention to the presence or absence of a newline character in the result array. Using \texttt{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 \texttt{fgets()} (along with any needed processing based on newline characters) instead of \texttt{gets_s().}

\textbf{Dynamic Allocation Functions}

ISO/IEC TR 24731-2 describes the \texttt{getline()} function derived from POSIX. The behavior of the \texttt{getline()} function is similar to that of \texttt{fgets()} but offers several extra features. First, if the input line is too long, rather than truncating input, the function resizes the buffer using \texttt{realloc()}. Second, if successful, it
Strings 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()**

```c
01  int ch;
02  char *p;
03  size_t buffer_size = 10;
04  char *buffer = malloc(buffer_size);
05  ssize_t size;
06
07  if ((size = getline(&buffer, &buffer_size, stdin)) == -1) {
08    /* handle error */
09  } else {
10    p = strchr(buffer, '\n');
11    if (p) {
12      *p = '\0';
13    } else {
14      /* newline not found, flush stdin to end of line */
15      while (((ch = getchar()) != '\n')
16          && !feof(stdin)
17          && !ferror(stdin)
18          );
19    }
20  }
21
22  /* ... work with buffer ... */
23
24  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.
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.

**Table 2.4 Alternative Functions for gets()**

<table>
<thead>
<tr>
<th>Function</th>
<th>Standard/TR</th>
<th>Retains Newline Character</th>
<th>Dynamically Allocates Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>fgets()</td>
<td>C99</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>getline()</td>
<td>TR 24731-2</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>gets_s()</td>
<td>C11</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

strncpy() and strcat()

The strncpy() 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 strncpy() 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**

1. dest = (char *)malloc(strlen(source) + 1);
2. if (dest) {
3.    strcpy(dest, source);
4. } else {
5.    /* handle error */
6.    ...
7. }

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 strncpy(). In other cases, it may still be desirable to replace the
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:

```c
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:

```c
1  errno_t strcpy_s(
2      char * restrict s1, rsize_t s1max, const char * restrict s2
3    );
```
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
01  errno_t strcpy_s(
02    char * restrict s1,
03    rsize_t s1max,
04    const char * restrict s2
05  ) {
06    errno_t   rc = -1;
07    const char  *msg;
08    rsize_t   s2len = strnlen_s(s2, s1max);
09    // Verify runtime constraints
10    if (nullptr_msg(msg, s1) && // s1 not NULL
11      nullptr_msg(msg, s2) && // s2 not NULL
12      maxsize_msg(msg, s1max) && // s1max <= RSIZE_MAX
13      zero_msg(msg, s1max) && // s1max != 0
14      a_gt_b_msg(msg, s2len, s1max - 1) &&
15          // s1max > strnlen_s(s2, s1max)
16      overlap_msg(msg,s1,s1max,s2,s2len) // s1 s2 no overlap
17    ) {
18      while (*s1++ = *s2++);
19      rc = 0;
20    } else {
21      // Runtime constraints violated, make dest string empty
22      if ((s1 != NULL) && (s1max > 0) && lte_rsizmax(s1max)) {
23          s1[0] = NULLCHAR;
24      }
```
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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Standard/TR</th>
<th>Buffer Overflow Protection</th>
<th>Guarantees Null Termination</th>
<th>May Truncate String</th>
<th>Allocates Dynamic Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>_strcpy()</td>
<td>C99</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>_strncpy()</td>
<td>C99</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>_strlcpy()</td>
<td>OpenBSD</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>_strdup()</td>
<td>TR 24731-2</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>_strcpy_s()</td>
<td>C11</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 2.6 summarizes some of the alternative functions for \texttt{strcat()} described in this section. TR 24731-2 does not define an alternative function to \texttt{strcat()}.

\begin{tabular}{ |c|c|c|c|c|c| }
\hline
 & Standard/TR & Buffer Overflow Protection & Guarantees Null Termination & May Truncate String & Allocates Dynamic Memory \\
\hline
\texttt{strcat()} & C99 & No & No & No & No \\
\hline
\texttt{strncat()} & C99 & Yes & No & Yes & No \\
\hline
\texttt{strlcat()}} & OpenBSD & Yes & Yes & Yes & No \\
\hline
\texttt{strcat_s()} & C11 & Yes & Yes & No & No \\
\hline
\end{tabular}

\textbf{strncpy()} and \textbf{strncat()}

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

The \texttt{strncpy()} library function performs a similar function to \texttt{strcpy()} but allows a maximum size \texttt{n} to be specified:

\begin{verbatim}
1  char *strncpy
2    char * restrict s1, const char * restrict s2, size_t n
3  );
\end{verbatim}

The \texttt{strncpy()} function can be used as shown in the following example:

\begin{verbatim}
strncpy(dest, source, dest_size - 1);
dest[dest_size - 1] = '\0';
\end{verbatim}

Because the \texttt{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 \texttt{strncpy()} function is frequently recommended as a “more secure” alternative to \texttt{strcpy()}. However, \texttt{strncpy()} is prone to string termination errors, as detailed shortly under “C11 Annex K Bounds-Checking Interfaces.”
The `strncat()` function has the following signature:

```c
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()`:

```c
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()`:

```c
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:

```c
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:

```c
1 char src1[100] = "hello";
2 char src2[7] = {'g','o','o','d','b','y','e'};
3 char dst1[6], dst2[5], dst3[5];
4 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 \( r_2 \) and the sequence \texttt{good\0} to \( \text{dst2} \):

\[
r_2 = \	ext{strncpy\_s}(\text{dst2}, \text{sizeof}(\text{dst2}), \text{src2}, 4);
\]

However, there is inadequate space to copy the \texttt{src1} string to \( \text{dst3} \). Consequently, if the following call to \texttt{strncpy\_s()} returns, \( r_3 \) is assigned a non-zero value and \( \text{dst3}[0] \) is assigned '\0':

\[
r_3 = \	ext{strncpy\_s}(\text{dst3}, \text{sizeof}(\text{dst3}), \text{src1}, \text{sizeof}(\text{src1}));
\]

If \texttt{strncpy()} had been used instead of \texttt{strncpy\_s()}, the destination array \( \text{dst3} \) would not have been properly null-terminated.

The \texttt{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 \texttt{strncat\_s()} function has the following signature:

1. \texttt{errno\_t strncat\_s(}
2. \hspace{1em} \texttt{char * restrict s1,}
3. \hspace{1em} \texttt{rsize\_t s1max,}
4. \hspace{1em} \texttt{const char * restrict s2,}
5. \hspace{1em} \texttt{rsize\_t n}
6. \texttt{);}

A runtime-constraint violation occurs and the \texttt{strncat\_s()} function returns a nonzero value if either the source or destination pointer is \texttt{NULL} or if the maximum length of the destination buffer is equal to 0 or greater than \texttt{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 \texttt{strncat\_s()} function also ensures null termination of the destination string.

The \texttt{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:

```c
1  strcat_s(
2    dest,
3    sizeof dest,
4    src,
5    (sizeof dest) - strnlen_s(dest, sizeof dest) - 1
6  );
```

Although the OpenBSD function `strlcat()` is similar to `strcat()`, 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, \( n + 1 \) bytes 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 2.7 Truncating Copy Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Standard/TR</th>
<th>Buffer Overflow Protection</th>
<th>Guarantees Null Termination</th>
<th>May Truncate String</th>
<th>Allocates Dynamic Memory</th>
<th>Checks Runtime Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strncpy()</code></td>
<td>C99</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>strlcpy()</code></td>
<td>OpenBSD</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>strndup()</code></td>
<td>TR 24731-2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>strncpy_s()</code></td>
<td>C11</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2.8 Truncating Concatenation Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Standard/TR</th>
<th>Buffer Overflow Protection</th>
<th>Guarantees Null Termination</th>
<th>May Truncate String</th>
<th>Allocates Dynamic Memory</th>
<th>Checks Runtime Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strncat()</code></td>
<td>C99</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>strlcat()</code></td>
<td>OpenBSD</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>strncat_s()</code></td>
<td>C11</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**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 non-zero 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:

```c
#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

```c
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:

```c
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>
<thead>
<tr>
<th>Value of type Argument</th>
<th>Operates on</th>
<th>If Unknown, Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Maximum object</td>
<td>(size_t) -1</td>
</tr>
<tr>
<td>1</td>
<td>Minimum object</td>
<td>(size_t) -1</td>
</tr>
<tr>
<td>2</td>
<td>Maximum object</td>
<td>(size_t) 0</td>
</tr>
<tr>
<td>3</td>
<td>Minimum object</td>
<td>(size_t) 0</td>
</tr>
</tbody>
</table>
**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() strncmp() 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:

```
1  __attribute__((nothrow)) memcpy(  
2     void * __restrict __dest,  
3     __const void * __restrict __src,  
4     size_t __len  
5  ) {  
6     return __memcpy_chk(  
7         __dest, __src, __len, __builtin_object_size(__dest, 0)  
8     );  
9  }
```

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

1. The following case is known to be correct:
   ```
   1  char buf[5];  
   2  memcpy(buf, foo, 5);  
   3  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:
   ```
   1  memcpy(buf, foo, n);  
   2  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 \texttt{\_\_memcpy\_chk()} or \texttt{\_\_strcpy\_chk()} are used in this case; these functions check whether buffer overflow happened. If buffer overflow is detected, \texttt{\_\_chk\_fail()} is called and typically aborts the application after writing a diagnostic message to \texttt{stderr}.

3. The following case is known to be incorrect:
   
   
   
   
   
   
   
   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:

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

\textbf{Learn More: Using \texttt{\_\_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:

\begin{verbatim}
01 char dest[BUFFER_SIZE];
02 char *src = /* valid pointer */;
03 size_t src_end = \_\_builtin\_object\_size(src, 0);
04 if (src_end == (size_t) -1 && /* don't know if src is too big */
05 strlen(src) < BUFFER_SIZE) {
06 strcpy(dest, src);
07 } else if (src_end <= BUFFER_SIZE) {
08 strcpy(dest, src);
09 } else {
10 /* src would overflow dest */
11 }
\end{verbatim}

The advantage of using \texttt{\_\_builtin\_object\_size()} is that if it returns a valid size (instead of 0 or \texttt{-1}), then the call to \texttt{strlen()} at runtime is unnecessary and can be bypassed, improving runtime performance.

GCC implements \texttt{strcpy()} as an inline function that calls \texttt{\_\_builtin\_strcpy\_chk()} when \texttt{_FORTIFY\_SOURCE} is defined. Otherwise, \texttt{strcpy()} is an ordinary glibc function. The \texttt{\_\_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:

```c
#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:

```c
#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.
2.6 Runtime Protection Strategies

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:

```cpp
#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 \texttt{-fstack-protector} and \texttt{-fno-stack-protector} options enable and disable stack-smashing protection for functions with vulnerable objects (such as arrays). The \texttt{-fstack-protector-all} and \texttt{-fno-stack-protector-all} options enable and disable the protection of every function, not just the functions with character arrays. Finally, the \texttt{-Wstack-protector} option emits warnings about functions that receive no stack protection when \texttt{-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 \texttt{struct} members so that a stack object of a type such as

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

would remain unprotected.
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
Strings 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:

```c
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:

```c
VOID 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.

```c
1  char *substring_before(char *s, size_t n, char c) {
2    for (int i = 0; i < n; ++i)
3      if (s[i] == c) {
4        s[i] = '\0';
5        return s;
6      }
7    s[0] = '\0';
8    return s;
9  }
```
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.


**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:


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.
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.

A
ABA problem, 393–398
ABI (application binary interface), 127–128
Absolute path name, 405–406, 432
Accelerated Requirements Method (ARM), 484
Access control lists (ACLs), 413
Access right(s), analysis and reduction, 494–495
ACLs. See Access control lists (ACLs)
ActiveX controls, vulnerabilities in, 515
Addition operations, 260–267
   one’s complement, 233
Address space layout randomization (ASLR), 111–116
Adve, Vikram, 506
AHP (Analytical Hierarchical Process), 485
AIR. See As-if infinitely ranged (AIR) integer model
Alert TA06-081A, 428
alloca() function, 146, 148–149, 153
   return values on success and error, 217, 217t
Alignment(s)
   definition, 147
   extended, 148
   fundamental, 148
   stronger/stricter, 148
   weaker, 148
alloca() function, 149–150
Allocation function(s), 163–168
   for array types, 163
   and deallocation functions, correct pairings, 176, 176t
   failure, 164–168, 172
   incorrect pairing of C and C++ allocation and deallocation functions and, 172–173
   for nonarray types, 163
Amdahl’s law, 361, 362f
American National Standards Institute (ANSI) C Standard, 20
   X3J11 committee, 19–20
Analytical Hierarchical Process (AHP), 485
The Annotated C++ Reference Manual (Ellis and Stroustrup), 20
ANSI. See American National Standards Institute (ANSI)
Apple file system forks, and equivalence errors, 436–437
Application binary interface (ABI), 127–129
Application Verifier, 222
Arbitrary memory write, 124–125, 127
   and atexit() function, 133–134
   and .dtors section, 129–131
Arbitrary memory write (continued)
  and global offset table, 127–129
  and longjmp() function, 134–136
  and structured exception handling, 138–139
  and system default exception handling, 139
  and virtual pointers, 133
Arbitrary write condition, 288
Arbitrary-precision arithmetic, 227, 292–293
Arc injection, 64, 69–70
Architecture and design, in software development, 486–503
Arena(s), jemalloc, 216
Argument(s), 309–310
  command-line, 43–44
  direct access to, 335–337
  malicious, 64
  naming, 313
  passing, 313
  sequentially ordered, 312–313, 312
  variable number of, 309–312
Argument pointer(s), 323, 323
  advancing, 324–325
  and buffer expansion, 346
  and variadic function implementation, 344–345
  moving, 324–325
Ariane 5 launcher, 301
Arithmetic. See also Integer(s)
  arbitrary-precision, 227, 292–293
  C language solution, 293
  bignum, 227
  GMP (GNU Multiple-Precision Arithmetic Library), 292
  Java BigInteger, 292
  modulo (modwrap semantics), 302
  one's complement, 233
  pointer, 260, 304
  usual integer conversions, 249
Arithmetic operations
  addition, 260–267
  one's complement, 233
  division and remainder, 274–279
  multiplication, 269–274
  subtraction, 267–269
ARM (Accelerated Requirements Method), 484
Arrays, 30
  character, 30
  count, 40
  fixed-length, and data from unbounded sources, 43
  length, 40
  scalars and, 174–175
  size, 31–32, 40
  variable-length (VLAs), 150–151
The Art of Computer Programming (Knuth), 181–182
As-if infinitely ranged (AIR) integer model, 303–304
  and observation point, 303–304
As-if rule, 369
ASLR (address space layout randomization), 111–116
asprintf() function, 340
atexit() function, 133–134
ATM, application-specific misuse case, 485, 486
Atomic operations, 376–378
  relaxed, 371
  use, 463
Attack surface
  analyzing, 494
  reducing, 494–495
  review, 516–517
Attack Surface Analyzer, 517
Attackers, definition, 14
AusCERT, 28
  Advisory AA-2000.02, 348
Automated teller machine (ATM), application-specific misuse case, 485, 486
Autovectorization, 358–359
Avoidance strategy(ies). See Mitigation(s)

B
Basic character set, 32
Basic Combined Programming Language (BCPL), 19
Basic Fuzzing Framework (BFF), 514
basic_string, 36–37
basic_string class, 80–81
mistakes using, 81–83
BCPL. See Basic Combined Programming Language (BCPL)
Best-fit memory allocation, 181
BFF (Basic Fuzzing Framework), 514
Black-box fuzzing, 513–514
Blacklisting, 501–502
Blaster worm, 1–5, 2f, 117
flawed logic exploited by, 5, 5f
Block devices, 407
Boehm-Demers-Weiser conservative garbage
collector, 169
Bound, definition, 30
Boundary(ies), exploitable, 500–501, 501f
Boundary tags, 181, 181n, 201–202, 201f
Branching
conditional, 71–72
unconditional, 71–72, 72f
BSS segment, 123–124
Buffer overflow(s), 53–54, 53f, 70, 118–120. See
also String(s)
arc injection, 64, 69–70
code injection, 64–70
detection, 72, 101–102
dlmalloc
frontlink technique, 191–195
unlink technique, 185–191
formatted output functions and, 319–321
in heap, 185–191
frontlink technique, 191–195
unlink technique, 185–191
inadequately bounded loops and, 122–123
mitigation strategies, detection and recovery,
72, 101–102
prevention, 72, 102
RtlHeap, 202–207
secure recovery from, 72, 101–102
in stack segment, 59
vulnerabilities, 117–118
__builtin_object_size() function, 102–106
__builtin___strcpy_chk() function, 105–106
Butenhof, David, 368
C
C*, 20
C and C++
alternatives to, 25
descendants of, 20
history of, 19–20
and implementation-defined behavior, 22, 23
legacy code, 24
and locale-specific behavior, 21, 23
popularity, 17–18, 18t, 19t
portability, 23–24
security problems with, 21–24
standards, 20
and type safety, 24
and undefined behavior, 22–24
and unspecified behavior, 21–22
and vulnerabilities, 21
C11 Annex K bounds-checking interfaces,
73–76, 282, 340–341
C++ Coding Standards: 101 Rules, Guidelines, and
Best Practices (Sutter and Alexandrescu),
83
The C Programming Language (Kernighan and
Ritchie), 19, 181–182
C range error detector (CRED), 107–108
C runtime (CRT) library, in Win32, 197–198
C Standard, memory management functions,
146–147
calloc() function, 147, 152, 153–154, 173
and integer wraparound vulnerability, 284
return values on success and error, 217, 217t
Canary(ies)
Random XOR, 109
stack, 108–109, 140
Canonicalization, 439–442, 499–500
Case sensitivity, and equivalence errors, 436
Casts, 38
CDE ToolTalk, 348–349
__cdec1, 313
Center for Strategic and International Studies
(CSIS), list of significant cyber events, 10
Cerb CerbNG, concurrency vulnerabilities, 400
CERT Advisory
CA-1996-20, 428
CA-1996-24, 428
CA-1996-25, 428
CA-1997-05, 428
CA-1997-06, 118
CA-1997-09, 118
CA-2000-06, 118
CA-2000-13, 348
CA-2001-27, 349
CA-2002-33, 223
CA-2003-02, 223
CA-2003-07, 428
CA-2003-12, 428
CA-2003-16, 2
CA-2003-23, 428
The CERT C Secure Coding Standard (Seacord), 482–483, 510
“Arrays (ARR),” 30
ARR01-C, 31–32
ARR32-C, 150
DCL03-C, 273
DCL12-C, 292
DCL34-C, 366–367
ERR00-C, 76
ERR02-C, 88
ERR03-C, 75
ERR38-CPP, 179–180
EXP33-C, 151
EXP34-C, 155
FIO01-C, 429, 432
FIO02-C, 440
FIO03-C, 432, 455, 456
FIO04-C, 45, 53
FIO05-C, 450, 464
FIO15-C, 429
FIO30-C, 338
FIO32-C, 445
FIO33-C, 45
FIO34-C, 86
FIO35-C, 86
FIO37-C, 64
FIO43-C, 460
INT01-C, 290
INT06-C, 339
INT07-C, 240
INT13-C, 281
INT15-C, 244
INT30-C, 293–294, 296
INT31-C, 293, 296–297
INT32-C, 293, 297
INT34-C, 280
MEM03-C, 152
MEM04-C, 156, 159
MEM07-C, 152
MEM09-C, 151
MEM11-C, 153
MEM32-C, 153–154
MEM35-C, 156
MEM36-C, 149
MEM08-CPP, 172–173
MEM39-CPP, 176
MSC06-C, 152
MSC10-C, 33
MSC14-C, 264, 268
MSC23-C, 162
MSC34-C, 42
POS01-C, 467
POS35-C, 466
POS36-C, 426
POS37-C, 428
SIG30-C, 355
“Signals (SIG),” 279
STR00-C, 39
STR01-C, 73
STR07-C, 74, 282
STR30-C, 35
STR31-C, 39, 41, 76
STR32-C, 49
STR35-C, 43
STR36-C, 36
CERT Vulnerability Note, 11
VU#29823, 348
VU#159523, 154
VU#192038, 222
VU#210409, 433
VU#286468, 349
VU#542081, 223
VU#568148, 2
VU#595507, 349
VU#650937, 223
VU#866472, 224
CERT/CC and coding standards, 482–483
Insider Threat Center, 9
role in security training, 481
ROSE Checkers SourceForge, 305
vulnerabilities reported to, 11, 12, 18
Vulnerability Disclosure Policy, 9n
Chamber of Commerce, U.S., computer network, hacker penetration of, 10
Change state property, 363, 469
Channel(s), analysis and reduction, 494–495
char, 30, 35, 37–39
Character devices, 407
Character set
basic, 32
execution, 32
multibyte, 32. See also UTF-8
Character string literals, 34–36
Character strings, 29–41
Character types, 37–39
integer, 240–241
Checklists, for software development, 516
Check-use-check pattern, 463–466
chroot jail, 470, 487n
chroot() system call, 487, 487n
clear(), 31
close() function, 410–411, 411t
cmd.exe, 4
Code audits, 515
for integer range errors, 306
Code injection, 64–69, 70
CodeSonar, 506
COFF (common object file format), 207n
Common desktop environment (CDE), 348
Common object file format (COFF), 207n
Compass/ROSE tool, 506–507, 511
Competitive intelligence professionals, as threat, 9–10
Compilation flags, 503–504, 504f
Compiler(s), 26–27
security features, 503–505
Compiler checks, 342–343
Compiler optimization, undefined behaviors in C and, 23
Compiler reordering, and thread safety, 369–370
Compiler-generated runtime error checks, 106, 300–301
Complete mediation, 488–489, 490f
Complete object, 148
Computer security, 12
Concatenating strings, 43–47
Concatenation functions, 89–93, 93t
truncating, 93–99, 99t
Concurrency
and ABA problem, 393–398
deadlocks, 385–391, 462
definition, 355
interleaved, 355, 356f
livelock, 385
and lock contention, 383, 392–393
mitigation pitfalls, 384–398
mitigation strategies, 368–384
atomic operations, 376–378
concurrent code properties, 383–384
data races, 370–371
happens before, 371
immutable data structures, 383
lock guards, 375
lock-free approaches, 379–380
memory barriers (fences), 378–379
memory model, 368–370
message queues, 380
mutexes, 374–375
reentrant functions, 383–384
relaxed atomic operations, 371
semaphores, 379
synchronization primitives, 371–374
thread safety, 383–384
parallel, 355, 356f
and parallelism, 355–359
and prematurely releasing a lock, 391–392
programming for, common errors, 362–368
corrupted values, 364–365
race conditions, 362–364
volatile objects, 365–368
single-threaded programs and, 354–355
spinlocks, 398
starvation, 385
vulnerabilities, 399–401
DoS attacks in multicore DRAM systems, 399
in system call wrappers, 400–401
time-of-audit-to-time-of-use (TOATTOU), 401
time-of-check-to-time-of-use (TOCTTOU), 401
time-of-replacement-to-time-of-use (TORTTOU), 401
Concurrent Versions System (CVS). See CVS
Concurrent-C, 20
Conforming program, 23
Conover, Matt, 198
const char, 35
Constructor attribute, 129–131
Container virtualization, 470
Contention, 383, 392–393
Control flow(s)
trusted, 450–451
untrusted, 450–451
Control transfer instructions, 125
Conversion specification, 314–315
Conversion specifier(s), 315, 315t–316t
Conversions, integer. See Integer conversions
Copy functions, string, 89–92, 92t
truncating, 93–99, 99t
Copying strings, 43–47
Costs
of Blaster worm, 4
of cybercrime, 6–7, 7t–8t
Counted minus loop, 290
Countermeasure(s). See Mitigation strategy(ies)
_countof(array), 40
equal, 343–344
Crackers. See also Attackers
definition, 9
CRED (C range error detector), 107–108
Crimes. See Cybercrime
Criminals. See also Attackers
as threat, 9
Critical section, 363
Critical undefined behavior, 303–304
CSIS. See Center for Strategic and International Studies (CSIS)
ctors section, 130
CVS buffer overflow vulnerability, 222
CVS server double-free, 214, 223–224
Cybercrime
costs of, 6–7, 7t–8t
traditional crimes becoming, 6, 8t
trends in, 6
underreporting of, 6
unnoticed, 6
unreported, 6
CyberSecurity Watch Survey, 2010, 6
Cyberterrorism, 10
Cyclone, 25
Cygwin, 25, 25n

D
D programming language, 25
DAG. See Directed acyclic graph (DAG)
Data
ad hoc, processing, 498
capsulation, 497
external, trusted vs. untrusted, 50
input validation and, 497–498
locations, 122–123
in nonstandard formats, processing, 498
sanitization, 500. See also Blacklisting;
Whitelisting
specifications for, 497
tainted, 51–52
Data execution prevention (DEP), 114–115
Data model(s), 241, 241f
Data parallelism, 357–359, 357f
Data pointer(s), 121, 124–125
Data races, 370–371. See also Deadlocks
Data streams, 408
Deadlocks, 385–391, 462
Deallocation function(s), 163, 164, 168–169
and allocation functions
correct pairings, 176, 176t
incorrect pairing, 172–173
for array types, 163
for nonarray types, 163
throwing an exception, 179–180
decode_pointer() function, 140–142
Defect report (DR), 400, 161–162
Defense in depth, 72, 120, 511–512
Defensive strategy(ies). See Mitigation strategy(ies)
Déjà vul, 152
Dekker’s example, 369, 378
delete expression, 162, 172–173
Denial-of-service (DoS), 4, 4n
DEP (data execution prevention), 114–115
Department of Homeland Security, Software Assurance Curriculum Project, 481
Destructor attribute, 129–131
Detection and recovery strategy(ies)
buffer overflow, 72, 101–102
runtime protection, 111–113
Development, software. See Software development
Development platforms, 25–27
Device files, 407, 445–448
preventing operations on, 445–448
Dhurjati, Dinakar, 506
Direct access, to arguments, 335–337
Directed acyclic graph (DAG), 404
Directory(ies), 405, 406
secure, 429, 470
shared, 458–461
Directory traversal, 432–435
vulnerable products, 434, 434f
Division operations, 274–279
dm alloc, 182–191
allocated and free memory chunks, structure, 182–183, 183f
buffer overflow
frontlink technique, 191–195
unlink technique, 185–191
double-free vulnerabilities, 191–195
free list double-linked structure, 183–184, 183f
<table>
<thead>
<tr>
<th>Concept</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>unlink() macro</td>
<td>184, 185f</td>
</tr>
<tr>
<td>writing to freed memory</td>
<td>195–196</td>
</tr>
<tr>
<td>DoS. See Denial-of-service (DoS)</td>
<td></td>
</tr>
<tr>
<td>Double-free vulnerability(ies), 157, 158, 160, 177–178. See also CVS server double-free</td>
<td></td>
</tr>
<tr>
<td>dlmalloc, 191–195</td>
<td></td>
</tr>
<tr>
<td>RtlHeap, 208–211</td>
<td></td>
</tr>
<tr>
<td>DRAM. See Dynamic random-access memory (DRAM) systems</td>
<td></td>
</tr>
<tr>
<td>Dranzer tool, 515, 515n</td>
<td></td>
</tr>
<tr>
<td>.dtors section, 129–131</td>
<td></td>
</tr>
<tr>
<td>Dynamic analysis, in race condition detection</td>
<td>471</td>
</tr>
<tr>
<td>Dynamic memory allocator, 146</td>
<td></td>
</tr>
<tr>
<td>Dynamic memory management, 145–224</td>
<td></td>
</tr>
<tr>
<td>aligned_alloc() function</td>
<td>146, 148–149, 153</td>
</tr>
<tr>
<td>alignment, 147–149</td>
<td></td>
</tr>
<tr>
<td>alloc() function, 149–150</td>
<td></td>
</tr>
<tr>
<td>allocation functions, 146–147</td>
<td></td>
</tr>
<tr>
<td>best-fit allocation, 181</td>
<td></td>
</tr>
<tr>
<td>C++, 162–172</td>
<td></td>
</tr>
<tr>
<td>common errors, 172–180</td>
<td></td>
</tr>
<tr>
<td>handling of allocation failures, 172</td>
<td></td>
</tr>
<tr>
<td>calloc() function, 147, 152–154</td>
<td></td>
</tr>
<tr>
<td>common errors, 151–162, 172–180</td>
<td></td>
</tr>
<tr>
<td>checking return values, 153–155</td>
<td></td>
</tr>
<tr>
<td>dereferencing null or invalid pointers, 155–156</td>
<td></td>
</tr>
<tr>
<td>DR #400, 161–162</td>
<td></td>
</tr>
<tr>
<td>freeing memory multiple times, 157–158</td>
<td></td>
</tr>
<tr>
<td>improperly paired functions, 172–176</td>
<td></td>
</tr>
<tr>
<td>initialization, 151–153</td>
<td></td>
</tr>
<tr>
<td>memory leaks, 158</td>
<td></td>
</tr>
<tr>
<td>referencing freed memory, 156–157</td>
<td></td>
</tr>
<tr>
<td>scalars and arrays, 174–175</td>
<td></td>
</tr>
<tr>
<td>zero-length allocations, 159–160</td>
<td></td>
</tr>
<tr>
<td>consistent conventions for, 212–213</td>
<td></td>
</tr>
<tr>
<td>first-fit allocation, 181</td>
<td></td>
</tr>
<tr>
<td>free() function, 147</td>
<td></td>
</tr>
<tr>
<td>improperly paired memory management functions and, 172–176</td>
<td></td>
</tr>
<tr>
<td>incorrect pairing of C and C++ allocation and deallocation functions and, 172–173</td>
<td></td>
</tr>
<tr>
<td>incorrect pairing of scalar and array operators and, 174–175</td>
<td></td>
</tr>
<tr>
<td>malloc function, 146–147</td>
<td></td>
</tr>
<tr>
<td>mitigation strategies, 212–222</td>
<td></td>
</tr>
<tr>
<td>notable vulnerabilities, 222–224</td>
<td></td>
</tr>
<tr>
<td>and randomization, 215</td>
<td></td>
</tr>
<tr>
<td>realloc() function, 146, 149, 153</td>
<td></td>
</tr>
<tr>
<td>Dynamic random-access memory (DRAM) systems</td>
<td></td>
</tr>
<tr>
<td>Dynamic randomized-input functional testing, 513–514</td>
<td></td>
</tr>
<tr>
<td>Dynamic storage allocation, 181–182</td>
<td></td>
</tr>
<tr>
<td>Dynamic storage duration, 162</td>
<td></td>
</tr>
<tr>
<td>Dynamic use of static content, 338–339</td>
<td></td>
</tr>
</tbody>
</table>

**E**

ECLAIR, 506, 506n

Economy of mechanism, 488–489

e-crime. See Cybercrime

Edison Design Group (EDG) compiler, 507

Education

online secure coding course, 481

in secure coding, 480–481

Effective C++ (Meyers), 341

Effective group ID (EGID), 416–427

Effective user ID (EUID), 415–427

Eiffel, 20

eip register. See Instruction pointer (eip)

ELF (executable and linking format), 127–129

encode_pointer() function, 140–142

Environment(s), supervised, 496

Equivalence errors, 435–437

Error conditions

concurrency programming, 362–368

corrupted values, 364–365

race conditions, 362–364

volatile objects, 365–368

dynamic memory management, 151–162, 172–180

checking return values, 153–155

dereferencing null or invalid pointers, 155–156

DR #400, 161–162

freeing memory multiple times, 157–158

improperly paired functions, 172–176

initialization, 151–153

memory leaks, 158

referencing freed memory, 156–157

scalars and arrays, 174–175

zero-length allocations, 159–160

Environment(s), supervised, 496

Economy of mechanism, 488–489

e-crime. See Cybercrime

Education

online secure coding course, 481

in secure coding, 480–481

Effective C++ (Meyers), 341

Effective group ID (EGID), 416–427

Effective user ID (EUID), 415–427

Eiffel, 20

eip register. See Instruction pointer (eip)

ELF (executable and linking format), 127–129

encode_pointer() function, 140–142

Environment(s), supervised, 496

Equivalence errors, 435–437

Error conditions

concurrency programming, 362–368

corrupted values, 364–365

race conditions, 362–364

volatile objects, 365–368

dynamic memory management, 151–162, 172–180

checking return values, 153–155

dereferencing null or invalid pointers, 155–156

DR #400, 161–162

freeing memory multiple times, 157–158

improperly paired functions, 172–176

initialization, 151–153

memory leaks, 158

referencing freed memory, 156–157

scalars and arrays, 174–175

zero-length allocations, 159–160
Error conditions (continued)
integers, 242t, 253t–256t. See also Integer
overflow
cconversion errors, 285, 288
exexceptional condition errors, 256–257,
257t
integer type range errors, 288
nonexceptional integer logic errors,
287–288
sign errors, 251, 254
truncation errors, 251, 254, 256–257,
257t, 259–260, 285–287, 288
string manipulation, 42–50
null-termination errors, 48–49
off-by-one errors, 47
string truncation, 49
unbounded string copies, 42–47
without functions, 49–50
Escape sequences, 34
EServ password-protected file access vulnerabil-
ity, 436
Ettercap version NG-0.7.2, 349
Evans, Jason, 216. See also jemalloc memory
manager
Event thread, 380
Exception, definition, 136
Exception handling, 136–139, 206
for new operator, 165
structured, 136–139
system default, 136–137, 139
vectored, 136–137
Exec Shield, 346
Executable and linking format (ELF), 127–129
cXecute Disable (XD) bit, 114
cXecute Never (XN) bit, 114
Execution character set, 32
“Exploiting Concurrency Vulnerabilities in Sys-
tem Call Wrappers” (Watson), 400
Exploits, 16–17
arc injection, 69–70
code injection, 64–69, 70
definition, 16
for IsPasswordOK program, stack smashing,
59–64
proof-of-concept, 16
remote procedure call, Blaster worm and, 3–4
return-oriented programming, 71–72
Extended alignment, 148
Extended base pointer (ebp) register, 56–57
Extended characters, 32
Extended integers, 226, 241
Extraction operator, 46–47
F
Fail-safe defaults, 488–489
Failure Observation Engine (FOE), 514, 514n
Fallon, Elias, 512
False negatives, in static analysis, 507–509, 508t
False positives, 304
in static analysis, 507–509, 508t
fastcall, 313
fchmod() function, 430–431
fclose() function, 410
fgets() function, 64, 84–86, 87, 89t
File(s)
attributes, 448–450
closing, 217
create without replace, 453–456
identification, 432–450
using multiple file attributes, 448–450
closing and opening, 409–410
c create delete, 444
special, 406–407, 445
c stream, 408
temporary
and appropriate privileges, 460, 461t
create without replace, 460, 461t
creation functions, 459–460, 461t
creation in shared directories, 459–460,
461t
and exclusive access, 460, 461t
and removal before termination, 460, 461t
with unique and unpredictable file names,
459–460, 461t
File I/O
access control, 413–432
changing privileges, 417–421
privilege management functions, 419–421
process privileges, 415–417
UNIX file permissions, 413–415
c basics of, 403–407
c byte input/output functions, 407
c in C++, 412
c concurrency, 467–469
advisory locks, 458
c exclusive locks, 458
c file locking, 436–458
Index

canonicalization, 439–442
unique and unpredictable, for temporary
files, 459–460, 461f
using file descriptors instead of, 468–469
File system(s), 404–406
distributed, 404
hierarchical, 404
Financial loss(es). See Costs
Finite-state automaton (FSA), 420, 420f
First-fit memory allocation, 181
Flags, 316
    compilation, 503–504, 504f
Floating point, 299–300, 324
fmemopen() function, 78–79
FOE (Failure Observation Engine), 514, 514f
foo() function, 57
    function epilogue for, 58–59
    function prologue for, 58, 58t
Foote, Jonathan, 514
fopen() function, 409–410, 411f
    and file creation, 453–456
    and permissions, 429–432
fopen_s() function, 456
Format string(s), 309–310, 314–318
    conversion specifications in, 314–315
    conversion specifier, 315, 315f–316t
    dynamic, 338–339
    excluding user input from, 338
    flags, 316
    interpretation, 314
    length modifier, 317, 317f–318t
    ordinary characters in, 314
    precision, 316
    width, 316
Format string vulnerability(ies), 319–320,
    349–351
brute-forcing, 351
    and crashing a program, 321–322
defeating stack randomization and, 332–333
detection, static taint analysis and, 343–344
and direct parameter access memory write,
    335–337
exploitable, 321, 349–351
heap-based, exploiting, 351
and viewing memory content, 324–326, 325f
and viewing stack content, 322–324, 323f
    wide-character, 332
    and writing addresses in two words, 334–335
WU-FTP, 319

mandatory locks, 458
named mutex object, 457–458
named semaphores, 457–458
shared locks, 458
synchronization primitives, 456–458
synchronizing across processes, 456–458
trusted/untrusted control flows, 450–451
data streams, 408
and exclusive access, 456–458
interfaces, 407–412
mitigation strategies, 461–471
    atomic operations, 463
    checking for symbolic links, 464–467
    chroot jail, 470
    closing the race window, 462–467
    container virtualization, 470
    controlling access to race object, 469–471
    dynamic analysis tools, 471
    eliminating race objects, 467–469
    exposure control, 470–471
    file descriptors versus file names, 468–469
    Helgrind tool, 471
    mutual exclusion migration, 462
    principle of least privilege, 469
    race detection tools, 471
    reopening files, 463–464
    secure directories, 470
    shared resources, 467–468
    static analysis tools, 471
    Thread Checker, 471
    thread-safe functions, 462–463
    and synchronizing across processes, 456–458
vulnerabilities
    directory traversal, 432–435
    path equivalence, 435–437
    privilege escalation, 418
    symlink-related, 438–439
    time of check, time of use (TOCTOU),
        451–453, 455
    wide-character input/output functions, 408,
        412
File lock, 458
    advisory, 458
    exclusive, 458
    mandatory, 458
    shared, 458
File name(s), 405–406
    binding to file objects, 432
Index

FormatGuard, 346–347
Formatted output, 309–351
  mitigation strategies, 337–348
    C11 Annex K bounds-checking interfaces, 340–341
    compiler checks, 342–343
    dynamic use of static content, 338–339
    excluding user input from format strings, 338
    Exec Shield, 346
    FormatGuard, 346–347
    iostream versus stdio, 341–342
    modifying variadic function implementation, 344–346
    restricting bytes written, 339–340
    static binary analysis, 347–348
    static taint analysis, 343–344
    testing, 342
    -Wformat flag, 343
    -Wformat-nonliteral flag, 343
    -Wformat-security flag, 343
  variadic functions, 309–313, 344–346
vulnerabilities
  buffer overflow, 319–321
  CDE ToolTalk, 348–349
  crashing a program, 321–322
  direct argument access, 335–337
  Ettercap version NG-0.7.2, 349
  internationalization, 331
  output streams, 321
  overwriting memory, 326–331
  viewing memory content, 324–326, 325f
  viewing stack content, 322–324, 323f
  Washington university FTP daemon, 348
  wide-character, 332
  writing addresses in two words, 334–335
Formatted output functions, 313–319
  and buffer overflow, 319–321
  exploiting, 319–332
  GCC implementation, 318
    limits, 318
  Visual C++ implementation, 318–319
    length modifier, 319
    limits, 319
    precision, 319, 319f
Forrester, Justin, 514
Fortify, 506
fprintf(), 314
Frame, definition, 56
Free lists, 198–200, 200f
FreeBSD, 214–215, 216
free() function, 152, 156–157, 162, 173, 181, 181n
fstat() function, 449–450
FTP session, directory traversal vulnerability, 433–434
Function(s). See specific function
Function epilogue, 58–59
Function pointer(s), 121, 123–124
  decoding, 140–142
  decryption, 140–142
  encoding, 140–142
  encryption, 140–142
Function prologue, 58
Fuzz testing, 513–515
fwrite() function, 39
G
Gadget(s)
  definition, 71
  return-oriented programming set of, 71–72, 71f
  Turing-complete set of, 71
Garbage collection, 169–172, 212
GCC (GNU Compiler Collection), 26–27, 506
  object size checking, 102–106
  security diagnostics, 507
“The Geometry of Innocent Flesh on the Bone” (Shacham), 72
getchar() function, 84–86
getdelim() function, 88
GetF11eType() function, 448
getline() function, 77, 87–89, 89t
gets() function, 42–43, 46, 51–53, 64, 84
  alternatives to, 84–89, 89t
gets_s() function, 86–87, 89t
Global offset table (GOT), 128–129
Gloger, Wolfram, 182
GLSA 200506-07, 349
GMP (GNU Multiple-Precision Arithmetic Library), 292
GNU Compiler Collection (GCC), 26–27, 506
  object size checking, 102–106
Index

555

GNU libc allocator, 182
GNU Multiple-Precision Arithmetic Library (GMP), 292
GOT (global offset table), 128–129
Group ID (GID), 413
/GS flag, 503–504, 504f
GSWK'T (Generic Software Wrappers Toolkit), concurrency vulnerabilities, 400
Guard pages, OpenBSD, 216
Guide to the Software Engineering Body of Knowledge (Bourque and Dupuis), 483–484
Guidelines, for software development, 516

H
Hackers
   politically motivated attacks by, 10
   as threat, 8–9
Hacker’s Delight (Warren), 299
Happens before, 371
Hard links, 442–445, 443f
   versus soft links, 444, 444f
Hazard pointers, 395–396
Heap exhaustion, 133–138
Heap memory
   randomization, in Windows, 113
   Win32 API, 197, 197f
Heap-based exploits, 146. See also Dynamic memory management
Heap-based vulnerabilities
   mitigation strategies, 212–222
   RtlHeap, 196–212
Helgrind tool, 471
Hi, definition, 30
Hocevar, Sam, 514, 514n
Horovitz, Oded, 198
Householder, Allen, 514–515
Howard, Michael, 298
HP Fortify Static Code Analyzer, 344
Hyperthreading, 354

I
IAT (import address table), 129
Implementation
   definition, 22
   and undefined behavior, 22–23
Import address table (IAT), 129
Independent security reviews, 516–517
Information warriors, as threat, 10
i-node, 405, 405f
   and hard links, 442–444, 444f
Input validation, 102, 497–498, 500, 518
Insiders, as threat, 9
Instruction pointer (eip), 57
   modifying, 125–127
Insure++, 221
int, 232
   minimum width, 237
int type, 38–39
Integer(s)
   character types, 240–241
   compiler- and platform-specific integral limits, 228, 228t
   data types, 226–246
      abstract, 291–292
      selection, 289–291
   definition, 225
   error conditions, 242t, 255t–256t. See also
      Integer overflow
      conversion errors, 285, 288
      exceptional condition errors, 256–257, 257t
   integer type range errors, 288
   nonexceptional integer logic errors, 287–288
   sign errors, 251, 254
   extended, 226, 241
int, 232
   minimum width, 237
intmax_t, 243–244
intptr_t, 245
long int, 232
   minimum width, 237
long long int, 232
   minimum width, 237
   mitigation strategies, 288–306
      abstract data types, 291–292
      arbitrary-precision arithmetic, 292–293
      as-if infinitely ranged (AIR) integer model, 303–304
      GCC -ftrapv flag, 300–301
Integer(s), mitigation strategies (continued)

- GNU Multiple-Precision Arithmetic Library (GMP), 292
- integer type selection, 289–291
- Java BigInteger, 292
- Microsoft Visual Studio c4244 warning, 305
- Microsoft Visual Studio runtime error checks, 106, 300
- modwrap semantics, 302
- overflow detection, 299–300
- postcondition testing, 297
- precondition testing, 295–297
- range checking, 288, 293–295
- restricted range usage, 302–303
- saturation semantics, 302
- secure integer libraries, 297–299
- source code audit, 306
- static analysis, 304–305
- testing, 305–306
- type safety, 292
- verifiably in-range operations, 301–303

One’s complement, 232, 233, 234–235, 235t

- operations, 256–283
  - addition, 260–267
  - assignment, 258–260
  - data parallelism and, 357–358
  - division and remainder, 274–279
  - error detection, 275–276
  - postcondition, 277–279
  - precondition, 276–277
  - and exceptional condition errors, 256–257, 257t
  - multiplication, 269–274
    - downcast from a larger type, 272–273
    - postcondition test using status flags, 270–272
    - precondition test, general, 273–274
    - shifts, 279–283
  - subtraction, 267–269
    - postcondition test using status flags, 267–268
  - verifiably in-range, 301–303

- operators
  - that can result in overflow, 239, 239t–240t
  - that can result in wrapping, 231, 231t
  - packed, 358
- platform-independent types for controlling width, 245
- platform-specific types, 245–246
  - ptdiff_t, 242–243
- range checking, 293–295
  - and integer wraparound vulnerability, 284–285
- representation, 226–227
  - comparisons of, 234–235, 235t
  - paddings bits, 226–227
  - precision, 227
  - width, 227, 237
  - rsize_t, 289–290

- security flaws involving, 225–226, 283
- shifts, 279–283
  - arithmetic (signed), 281, 281f
  - left shift, 279–281, 280f, 283
  - logical (unsigned), 281, 281f
  - right shift, 279, 281–282
  - short int, 232
    - minimum width, 237
  - sign and magnitude, 232, 234–235, 235t
  - signed, 231–235, 240–241
    - ranges, 235–237, 235t–236t, 236f
  - signed char, 232, 240
    - minimum width, 237
  - size_t, 242, 289–291
  - standard, 226, 232
  - truncation toward zero, 274
  - two’s complement, 232–233, 234–235, 234f, 234t, 235t, 239
    - and unary negation (–), 279
    - to unsigned conversion, 254, 255f
- typedefs, 241
  - uintmax_t, 243–244
  - uintptr_t, 245
  - unary negation (–), 279
  - unsigned, 227–229, 240–241
  - to two’s complement conversion, 251, 251f
  - unsigned char, 232, 240–241
- vulnerabilities, 283–288. See also Integer wraparound
  - conversion errors, 285
  - nonexceptional integer logic errors, 287–288
- truncation errors, 285–287

- Integer conversions, 246–256
  - explicit, 246
  - implicit, 246, 256
  - and loss of sign, 251, 254, 256
  - and loss of value, 251, 254, 256
promotions, 247–249
rank, 246–247
from signed types, 253–255, 255t–256t
loss of precision, 253, 255t–256t
to unsigned, 253–255, 255t–256t
from unsigned types, 250–253, 252t
loss of precision, 250, 252t
to signed, 250–252, 252t
usual arithmetic, 249
Integer overflow, 237–239, 239t–240t, 256–257, 257t, 261, 288
detection, 299–300
fussy, 294–295
signed
resulting from addition, 261–262
avoiding or detecting, 262–265
downcast from a larger type, 265
postcondition test using status flags, 263–264
precondition test, general, 264–265
precondition test, two’s complement, 264
resulting from division, 274
detecting and avoiding, 276–279
resulting from multiplication, 269
detecting or avoiding, 271–274
resulting from subtraction
avoiding or detecting, 268
precondition test, 268
Integer wraparound, 229–231, 256–257, 257t, 283–285
resulting from addition, 261
avoiding or detecting, 265–267
postcondition test, 266–267
using status flags, 265–266
precondition test, 266
resulting from multiplication, detecting or avoiding, 271–274
resulting from subtraction
avoiding or detecting, 269
postcondition test, 269
postcondition test using status flags, 269
precondition test, 269
Intellectual property, theft of, 9
Interface(s), exploitable, 500–501, 501f
Internationalization, formatted output vulnerability, 331
Internet Security Glossary, version 2, 483
Internet Security Systems Security Advisory, 349
Interprocess communication (IPC) mechanism(s), 459
intmax_t, 243–244
intptr_t, 245
I/O. See File I/O
iOS, ASLR (address space layout randomization), 116
iostream, 341–342
islower() function, 21
ISO/IEC
9899-1990, 20
9899:1999, 482
14882:2011, 20
24731, 74
TR 24731-1, 282, 299, 483
TR 24731-2, 76–77, 87–88, 92, 93, 99, 483
TS 17961 C Secure Coding Rules, 15, 217, 483, 509–510
conformance test suite for, 510
IsPasswordOK(), 51–53, 52f
security flaw in, 52–53, 53f, 59–64, 62f, 63f
istream class, 46
Iterators, 81
invalid, 81–82
J
Java, 25
Java BigInteger, 292
Java Native Interface (JNI), 25
jemalloc memory manager, 216–217
JIT. See Just-in-time (JIT) compiler
jmp_buf type, 134–136
Jones, Richard, 506
JPEG files, comment field, unsigned integer wraparound vulnerability, 283–284
Just-in-time (JIT) compiler, and W^X policy, 114–115
K
Kamp, Poul-Henning, 213. See also phkmalloc
Kelly, Paul, 506
Kerberos
buffer overrun vulnerability, 118
double-free vulnerabilities, 224
Klocwork, 506
Knuth, Donald, 181–182
K&R. See The C Programming Language

L
Lam, Monica, 506
Last Stage of Delirium (LSD) Research Group, 2
LDRA, 506
Lea, Doug, 146
memory allocator (dlmalloc), 182–191. See also dlmalloc
lea instruction, 65–66
Least common mechanism, 489, 492
Least privilege, 70, 489–492, 494
Legacy code, C and C++, 24
Lesk, M. E., 309n
libpng library, 155–156
Libsafe, 107
libsafe library, 496
Libverify, 107
Linux, 26
address space layout randomization, 112
file systems supported, 404
PaX patch, 112, 115–116
Livelock, 462
Lo, definition, 30
Load effective address (lea) instruction, 65–66
Locale, 32
Lock guards, 375
long int, 232
minimum width, 237
long long int, 232
minimum width, 237
longjmp() function, 134–136
Look-aside lists, 200, 200f, 212
LSD (Last Stage of Delirium Research Group), 2
malloc() function, 151–155, 173, 181
Manadhata, Pratyusa, 517
mbstowcs() function, 35
MDAC. See Microsoft Data Access Components (MDAC)
Memcheck, 219–221
memcpy() function, 39, 100
and object size checking, 104–105
memcpy_s() function, 100
memmove() function, 100
memmove_s() function, 100
Memory. See also Dynamic memory management
chunks, 201–202, 201f
double-free, 157, 158, 160
RtlHeap, 208–211
freed
accessing, 217
referencing, 156–157
writing to, dlmalloc, 195–196
writing to, RtlHeap, 207–208
freeing, 217
multiple times, 157–158, 176–179, 218
heap
randomization, in Windows, 113
Win32 API, 197, 197f
management modes, string-handling functions, 73
overwriting, 326–331
process, organization, 54, 55f
data declarations and, 123
read-only, 54
stack, randomization, in Windows, 113
uninitialized, referencing, 218
virtual, Win32 API, 196–197, 197f
zero-length allocations, 159–160
Memory fence(s), 368, 378–379
Memory leak(s), 158, 177
automatic detection of, 158
detection
Insure++, 221
Purify, 218
Valgrind tool, 221
Memory manager(s), 146, 180–182
memset() function, 152
memset_s() function, 152
Message queues, 380
Messier, Matt, 498
Metasploit Project, 3
Index

Meyers, Scott, 341
Microsoft Data Access Components (MDAC), buffer overflow vulnerability, 223
Microsoft Office, vulnerabilities in, SDL and, 474, 475f
Microsoft OpenOffice, vulnerabilities in, SDL and, 474, 475f
Microsoft Security Bulletin
  MS02-65, 223
  MS03-026, 2
Microsoft Visual Studio. See Visual Studio
Microsoft Windows. See Windows
Miller, Barton, 514
MIT krb5 library, 213
MIT krb5 Security Advisory 2004-002, 224
Mitigation(s), definition, 17
Mitigation pitfalls, concurrency, 384–398
Mitigation strategy(ies)
  applications, 474
  broad, 473
  buffer overflow, detection and recovery, 72, 101–102
  concurrency, 368–384
    atomic operations, 376–378
    concurrent code properties, 383–384
    data races, 370–371
    happens before, 371
    immutable data structures, 383
    lock guards, 375
    lock-free approaches, 379–380
    memory barriers (fences), 378–379
    memory model, 368–370
    message queues, 380
    mutexes, 374–375
    reentrant functions, 383–384
    relaxed atomic operations, 371
    semaphores, 379
    synchronization primitives, 371–374
    thread safety, 383–384
  dynamic memory management, 212–222
  file I/O, 461–471
    atomic operations, 463
    checking for symbolic links, 464–467
    chroot jail, 470
    closing the race window, 462–467
    container virtualization, 470
    controlling access to race object, 469–471
    dynamic analysis tools, 471
    eliminating race objects, 467–469
    exposure control, 470–471
    file descriptors versus file names, 468–469
    Helgrind tool, 471
    mutual exclusion migration, 462
    principle of least privilege, 469
    race detection tools, 471
    reopening files, 463–464
    secure directories, 470
    shared resources, 467–468
    static analysis tools, 471
    Thread Checker, 471
    thread-safe functions, 462–463
  formatted output, 337–348
    C11 Annex K bounds-checking interfaces, 340–341
    compiler checks, 342–343
    dynamic use of static content, 338–339
    excluding user input from format strings, 338
    Exec Shield, 346
    FormatGuard, 346–347
    ostream versus stdio, 341–342
    modifying variadic function implementation, 344–346
    restricting bytes written, 339–340
    static binary analysis, 347–348
    static taint analysis, 343–344
    testing, 342
    -Wformat flag, 343
    -Wformat-nonliteral flag, 343
    -Wformat-security flag, 343
  heap-based vulnerabilities, 212–222
  integers, 288–306
    abstract data types, 291–292
    arbitrary-precision arithmetic, 292–293
    as-if infinitely ranged (AIR) integer model, 303–304
    GCC -ftrapv flag, 300–301
    GNU Multiple-Precision Arithmetic Library (GMP), 292
    integer type selection, 289–291
    Java BigInteger, 292
    Microsoft Visual Studio C4244 warning, 305
    Microsoft Visual Studio runtime error checks, 106, 300
    modwrap semantics, 302
    overflow detection, 299–300
    postcondition testing, 297
Mitigation strategy(ies), integers (continued)
precondition testing, 295–297
range checking, 288, 293–295
restricted range usage, 302–303
saturation semantics, 302
secure integer libraries, 297–299
source code audit, 306
static analysis, 304–305
testing, 305–306
type safety, 292
verifiably in-range operations, 301–303
pointer subterfuge, 139–142
race conditions, 461
strings, 72–83
C11 Annex K bounds-checking interfaces, 73–76, 282, 340–341
C++ std::basic_string, 80–81
detection and recovery, 101–102
dynamic allocation functions, 76–80
input validation, 102
invalidating string object references, 81–83
object size checking, 102–106
runtime protection, 101–117
mkstemp function, secure and insecure use of, 461
mkstemp() function, 431–432
mktemp function, secure and insecure use of, 461
Mode(s), file opening, 409–410
Modula 3, 20
Moore, H. D., 3
Morris worm, 117
msblast.exe, 4
MTA. See Mail transfer agent (MTA)
Multibyte character set, 32. See also UTF-8
Multibyte string, 32
Multiplication operations, 269–274
Multithreading, 354–355, 368
Mutex(es), 374–375. See also Named mutex object

N
Named mutex object, 457–458
Named pipes, 407
Named semaphores, 457–458
NASA. See National Aeronautics and Space Administration (NASA)
National Aeronautics and Space Administration (NASA), advanced persistent threat attacks against, 10
National Institute of Standards and Technology (NIST), Static Analysis Tool Exposition (SATE), 509
National Vulnerability Database (NVD), vulnerabilities cataloged by, 11, 11f
Negative zero, 234
NEON instructions, 357
NetBSD Security Advisory 2000-002, 284
Network administrators, definition, 13
New expression, 162–163, 172–173, 175
incorrect use, 172
nothrow form, 172
New handler, 167–168
NIST. See National Institute of Standards and Technology (NIST)
No eXecute (NX) bit, 114
Normalization, 499–500
NTBS (null-terminated byte string), 36–37
NTMBS (null-terminated multibyte string), 36
Null character, 32, 34, 332
Null pointer(s), 212
Null-terminated byte string (NTBS), 36–37
Null-terminated multibyte string (NTMBS), 36
NVD. See National Vulnerability Database (NVD)
NX (No eXecute) bit, 114

O
Object pointer(s), 121, 124–125
Objective-C, 20
Obsolescent feature(s), 162
Off-the-shelf software, 495–496
on_exit() function, 133–134
Open design, 489, 490
OpenBSD, 215–216
security options for, 216, 216t
open() function, 410–411, 411t
and file creation, 453–456
and permissions, 429–431
open_memstream() function, 78
OpenSHH
privilege escalation vulnerability, 418
secure shell implementation, 487–488, 487f
open_wmemstream() function, 78
Operating system(s), 26
    and runtime protection strategies, 111–116
detection and recovery, 111–113
operator delete() function, 163, 164, 168–169, 173, 174
operator delete[]() function, 163, 168, 173–175
operator new, 162–163
operator new() function, 163, 164, 173–175
    and member new() function, failure to properly pair, 175
operator new[]() function, 163, 173–175
Out-of-bounds store, 304
__output() function, 318–319
Overaligned type, 148

P
Padding bits, 226–227
Page(s), in Win32 virtual memory API, 196
Parallelism, 355–359
    achievable, program structure and, 360, 360f
    Amdahl’s law, 361, 362f
    data, 357–359, 357f
    limits, 360, 361f
    and performance goals, 359–361
    task, 359, 359f
    and work-to-span ratio, 360, 361f
passwd program, 422
Path(s), canonical, 499–500
Path equivalence errors, 435–437
Path name(s), 405–406, 406f
    absolute, 405–406, 432
canonicalization, 439–442
relative, 406, 432, 435
resolution, 432
PCLint, 506
Penetration testing, 513
Permission(s)
    definition, 413
    management, 428–432
    on newly created files, 429–432
Pethia, Richard, 4
Phishing, 9
phkmalloc, 213–215. See also OpenBSD
    security implications, 214, 214t
ping program, 423–424
Placement new, 163
correct and incorrect use of, 175–176
PLT (procedure linkage table), 129
Pointer(s), 30, 31
data, 121, 124–125
disguised, and garbage collection, 169–170
function, 121, 123–124
decoding, 140–142
decryption, 140
coding, 140–142
cipher, 140
hazard, 395–396
invalid
dereferencing, 155–156
formed by library function, 218
to member type, 121
null, 212
dereferencing, 155–156
object, 121, 124–125
out-of-domain, dereferencing, 217
safely derived, 170
safety, management, 170–171
smart, 178–179
reference-counted, 178–179
to wide string, 34
Pointer arithmetic, 260, 304
Pointer subterfuge
    definition, 121
    mitigation strategies, 139–142
pointer_safety, 170
Portability, C and C++, 23–24
Portable executable (PE) file, 129, 207, 207n
“A Portable I/O Package” (Lesk), 309n
POSIX
    file descriptors, 410–411
    open and close file functions, 410–411
    threading library, 368
    umask process, 429–432, 430f
Preservation, and type safety, 24
Prevent, 506, 512
printf() function, 309, 314
Privilege(s)
    appropriate, 420
    changing, 417–421
definition, 413
dropping, 418, 425–426
    revocation order for, 426
    escalated, 418
    escalation, 418
    least, 489–492, 494
Privilege(s) (continued)
management, vulnerabilities associated with, 427–428
management functions, 419–421
managing, 422–428
process, 415–417
separation of, 489, 490
Procedure linkage table (PLT), 129
Process, definition, 54
Process environment block (PEB), 198, 199
Process group IDs, 416
Process memory, organization, 54, 55
data declarations and, 123
Process privileges, 415–417
Process user IDs, 415–416
Programmer, definition, 13
Programming language(s)
alternatives to C, 25
popularity
long-term trends in, 18, 19
TIOBE index of, 17–18, 18
Progress, and type safety, 24
Promotions, integer conversions, 247–249
ProPolice. See Stack-Smashing Protector (ProPolice)
Psychological acceptability, 489, 492–493
Pure binary notation, 39
Purify, 218–219, 512
PurifyPlus, 218–219
puts() function, 51

Q
Quality management, software development, 479–480
Quality requirements engineering, 483–485

R
Race conditions, 362–364, 450–461
canonicalization and, 441
change state property, 363, 469
and concurrency property, 363
detection
dynamic analysis tools, 471
static analysis tools, 471
using check-use-check pattern, 465–466
and exclusive access, 456–458
file-related, eliminating, 467–469
from GNU file utilities, 451
and shared directories, 458–461
and shared object property, 363
time of check, time of use (TOCTOU), 451–453, 455
vulnerabilities related to, mitigation strategies, 461
Race object
controlling access to, 469–471
eliminating, 467–469
Race window, 451
closing, 462–467
critical section, 363
definition, 363
identification, 363
RAII. See Resource Acquisition Is Initialization (RAII)
rand() function, 285
Random XOR canaries, 109
Range checking, integers, 293–295
Ranges of integers, 235–237, 235–236, 236
Read-only memory, 54
Real group ID (RGID), 416
Real user ID (RUID), 415–417
realToC, return values on success and error, 217, 217
realToC() function, 146, 149, 153, 159–162
realpath() function, 440–441, 495–496
Reentrant functions, 383–384
Reference-counted smart pointer(s), 178–179
Region(s), in Win32 virtual memory API, 196
Relative path name, 406, 432, 435
Remote login, 117–118
Remote procedure call (RPC), buffer overflow vulnerability, 2–3, 2n
Resource Acquisition Is Initialization (RAII), 165–166, 375
Resource-exhaustion attack, 158
Return-oriented programming, 71–72
Risk assessment. See Threat assessment
rlogin program, 117–118
ROSE, 304–305, 506–507
RPC (remote procedure call), buffer overflow vulnerability, 2–3, 2n
RTL (runtime linker), 129
RtlHeap, 146, 146n
buffer overflows, 202–207
data structures, 198–202
free lists, 198–200, 200f
look-aside lists, 200, 200f, 212
memory chunks, 201–202, 201f
process environment block, 198, 199f
double-free vulnerabilities, 208–211
heap-based vulnerabilities, 196–212
and writing to freed memory, 207–208
Runtime analysis tools, 218–222
Runtime bounds checkers, 106–108, 506
Runtime error checks
compiler-generated, 106, 300–301
GCC -ftrapv flag, 300–301
Microsoft Visual Studio, 106, 300
Runtime linker (RTL), 129
Runtime protection strategies, 101–117
advances in (future directions for), 116–117
operating system, 111–116
Runtime-constraint handler, 75–76, 299–300
RUS-CERT Advisory 2002-08:02, 284
Ruwase, Olatunji, 506

S
Safe-Secure C/C++ (SSCC), 116–117, 117f,
505–506, 507f
SAFE SEH, 138, 138n
Sanitization, 500. See also Blacklisting;
Whitelisting
Saved set-group-ID (SSGID), 416
Saved set-user-ID (SSUID), 415–416
SCADA (supervisory control and data acquisition), terrorist threat to, 10
Scalar registers, 357
SCALe (Source Code Analysis Laboratory), 510–511
Scott, Roger, 512
SDL. See Security Development Lifecycle (SDL)
Secunia Advisory SA15535, 349
Secure design patterns, 488
Secure wrappers, 496
Security
developmental elements, 12
independent reviews, 516–517
operational elements, 12
requirements, 481–486
Security analyst, definition, 13
Security concepts, 12–17, 13f
Security Development Lifecycle (SDL), 474–480,
474f, 505. See also Simplified SDL
Security flaw(s)
definition, 14
elimination of, 17
and vulnerabilities, 15
Security policy
definition, 14
explicit, 14
implicit, 14
Security quality requirements engineering
(SQUARE), 483–485
Security researcher, definition, 14
Security Tracker Alert ID 1014084, 349
Security training, 480–481
Security use/misuse cases, 485, 485l, 486t
SecurityFocus Bugtraq ID 1387, 348
SEH. See Structured exception handling (SEH)
Semaphores, 379. See also Named semaphores
Sendmail, vulnerabilities, 428
Separation of privilege, 489, 490
SESS (Summit on Education in Secure Software),
480–481
setegid() function, 419, 425
seteuid() function, 419–421
Setgid programs, 422
setgid() function, 425
setjmp() macro, 134–135
setlocale() function, 32
setresgid() function, 425
setresuid() function, 419, 425
setreuid() function, 419, 421
Setuid programs, 422–428
setuid() function, 419–428
Set-user-ID-root program, 422–424
Shacham, Hovav, 72
Shannon, Gregory E., 11
Shared directories, 458–461
Shellcode, 64
 injected, location of, 69
Shift state(s)
initial, 32
locale-specific, 32
short int, 232
 minimum width, 237
Shortcuts, 453
Signal(s), in management of division errors, 278–279
Signal handler(s), concurrency issues and, 354–355
*signed* char, 37–38, 232, 240–241
   minimum width, 237
Signed integer(s), 231–235, 240–241
   ranges, 235–237, 235t–236t, 236f
*Simplified Implementation of the Microsoft SDL*, 475
Simplified SDL, mapping of resources and tools to, 475, 475t–476t
Single instruction, multiple data (SIMD) computing, 148–149, 357
sizeof(array), 31, 40
sizeof operator, 31–32
sprintf() function, 340
Smart pointer(s), 178–179
ssprintf() function, 45, 314, 339–340
Sockets, 407
Software, off-the-shelf, 495–496
Software components, 12
Software defect(s), 14–15
   definition, 14
   per thousand lines of code, 27
   static analysis, 512
Software development
   architecture and design, 486–503
   blacklisting, 501–502
   code audits, 515
   data sanitization, 500
   defect removal in, 479–480
   defense in depth, 511–512
   fuzz testing, 513–515
   guidelines and checklists, 516
   implementation, 503–512
   independent security reviews in, 516–517
   input validation, 497–498
   penetration testing, 513
   planning, 477–479, 478f
   quality management, 479–480
   requirements, 481–486
   secure launch, 477–479, 478f
   secure wrappers, 496
   security principles, 488–493
      complete mediation, 488–489, 490f
      economy of mechanism, 488–489
      fail-safe defaults, 488–489
   least common mechanism, 489, 492
   least privilege, 489–492, 494
   open design, 489, 490
   psychological acceptability, 489, 492–493
   separation of privilege, 489, 490
   testing, 503
   threat modeling, 493–494
   tracking, 477–479, 478f
   trust boundaries, 498–501
   TSP-Secure, 477–480
   validation, 500
   verification, 512–517
   and vulnerabilities in existing code, 495–496
   whitelisting, 502–503
Software security, threats to, 11–12
Source code, 12–13
   audits, 515
   for integer range errors, 306
Source Code Analysis Laboratory (SCALe), 510–511
SourceForge, 511, 511n
Special files, 406–407, 445
Spies, corporate. See Competitive intelligence professionals
Spinlocks, 398, 457
Splint, 305
ssprintf() function, 43, 45–47, 77, 309, 314, 339–340
SQUARE. See Security quality requirements engineering (SQUARE)
sscanf() function, 77
SSCC. See Safe-Secure C/C++ (SSCC)
SSE. See Streaming SIMD Extensions (SSE)
SSP. See Stack-Smashing Protector (ProPolice)
StackShield, 143
Stack(s)
   and calling a subroutine, 55–56, 56f
   management, 55–59, 55f
   nonexecutable, 113
   randomization, 332–337
   defeating, 332–333
   Exec Shield and, 346
   smashing, 59, 60f, 61f. See also Stack-Smashing Protector (ProPolice)
   structure, 55, 55f
   Stack-Smashing Protector (ProPolice) and, 110, 111f
Stack buffer overrun detection, 108–109
Stack canaries, 108–109
Stack memory, randomization, in Windows, 113
Stack pointer, 57
StackGuard, 108, 109, 143
Stack-Smashing Protector (ProPolice), 108, 110, 111f
Standard library error, detection and handling of, 217
Standard template library (STL), checked implementation, 82
Standards, secure coding, 481–483
State-dependent encoding, 32
stat() function, 449–450
Static analysis, 217–218, 304–305
for format string vulnerabilities, 343–344
and implementation, 506–510
in race condition detection, 471
thread role analysis, 382–383
and verification, 512
Static assertion, 273
Static binary analysis, 347–348
-std flag, 27
std::bad_array_new_length, 166–167
std::basic_string, 36
__stdcall, 313
stdio, 341–342
std::stream class, 46
std::string class, 80–81
Sticky bit, 415
STL (standard template library), checked implementation, 82
Storage duration, 147, 162
allocated, 147
dynamic, 162
strcat() function, 43, 49, 73, 89, 93t, 94
strcat_s() function, 73, 90–92, 93t
strcpyp() function, 51–53
strcpy() function, 43–44, 48, 66–67, 67t–68t, 73, 89–90, 92t, 94
and object size checking, 104–105
strncpy_s() function, 73, 90–92, 92t
strdup() function, 45, 92, 92t
Stream
associated with memory buffer, 77–78
opening, to write to memory, 78–79
Stream files, 408
Streaming SIMD Extensions (SSE), 148–149, 357
Strictly conforming program, 23
String(s)
in C++, 36–37
concatenating, 43–47
concatenation functions, 89–93, 93t
copy functions, 89–92, 92t
copying, 43–47
data type, 30–32
definition, 30
error conditions, 42–50
null-termination errors, 48–49
off-by-one errors, 47
string truncation, 49
unbounded string copies, 42–47
without functions, 49–50
length, 30, 30f, 40
definition, 31
mitigation strategies, 72–83
C11 Annex K bounds-checking interfaces, 73–76, 282, 340–341
C++ std::basic_string, 80–81
detection and recovery, 101–102
dynamic allocation functions, 76–80
input validation, 102
invalidating string object references, 81–83
object size checking, 102–106
runtime protection, 101–117
multibyte, 32
null-terminated, 36–37, 48–49
definition, 31
pointer to, 30
sizing, 39–41
storage for, 76
symbolic verification technique (Yu et al.), 306
truncating concatenation functions, 93–99, 99t
truncating copy functions, 93–99, 99t
truncation, 49
value of, 30
vulnerabilities and exploits, 50–72, 117–118
wide, 33–34
sizing, 40–41
String class(es), 36–37
String literals, 34–36
String-handling functions, 73, 84–101
str1cat() function, 90, 93t, 98, 99t
str1cpy() function, 90, 92t, 96, 99t
strlnc() function, 31, 37, 40–41, 44, 48, 100–101
strncat() function, 49, 73, 93–95, 93t, 98, 99t
strncat_s() function, 73, 95, 97–98, 99t
strncpy() function, 48–49, 73, 90, 92t, 93–95, 96, 99t
strncpy_s() function, 73, 95–98, 99t
strndup() function, 99, 99t
strnlen() function, 101
strnlen_s() function, 100–101
strtok() function, 49
Structured exception handling (SEH), 136–139, 277–278
Subobject(s), 148
Subroutine, calling, 55–56, 56f
Summit on Education in Secure Software (SESS), 480–481
Sun tarball vulnerability, 152
Supervised environments, 496
Supervisory control and data acquisition (SCADA), terrorist threat to, 10
Supplementary group IDs, 416, 426–427
svchost.exe, 4
Symbolic links, 406, 437–439, 437f, 452–453
checking for, 464–467
symLink() system call, 437
SYN flooding, 4, 4n
syslog() function, 314
System administrator, definition, 13
System call wrappers, concurrency vulnerabilities, 400–401
System default exception handling, 136–137, 139
System integrator, definition, 13
Systrace, 496
  concurrency vulnerabilities, 400

T
Tainted value(s), 51
tar program, 152, 152n
tar utility, 152
Target(s), analysis and reduction, 494–495
Target size, definition, 31
Task parallelism, 359, 359f
tcp_wrappers package, 502–503
Team Software Process for Secure Software Development (TSP-Secure), 477–480
TEBs. See Thread environment blocks (TEBs)
Temporary file(s) and appropriate privileges, 460, 461t
create without replace, 460, 461t
creation functions, 459–460, 461t
creation in shared directories, 459–460, 461t
and exclusive access, 460, 461t
and removal before termination, 460, 461t
with unique and unpredictable file names, 459–460, 461t
Terrorists. See also Attackers as threat, 10
Thread Checker, 471
Thread environment blocks (TEBs), 198
Thread role(s), 381, 381n
Thread role analysis, 380–383
  annotation language, 381–382
  static analysis, 382–383
Thread safety, 368–370, 383–384
Thread support, 368
Thread usage policies, 380–381
Thread-safe functions, 462–463
Threat(s)
  competitive intelligence professionals as, 9–10
  criminals as, 9
  definition, 8
  hackers as, 8–9
  information warriors as, 10
  insiders as, 9
  to software security, 11–12
  terrorists as, 10
Threat assessment, 5–12
Threat modeling, 493–494
Threat Modeling Tool, 494, 494n
Time of check, time of use (TOCTOU), 401, 451–453, 455
Time-of-audit-to-time-of-use (TOATTOU), 401
Time-of-check-to-time-of-use (TOCTTOU), 401
Time-of-replacement-to-time-of-use (TORTTOU), 401
TIOBE index, 17–18, 18t
TIS. See Tool Interface Standards committee (TIS)
tmpfile function, secure and insecure use of, 461t
tmpfile_s function, secure and insecure use of, 461t
tmpnam function, secure and insecure use of, 461t
Use/misuse cases, 485, 485f, 486f
User ID (UID), 413
User name, 413
UTF-8, 32–33
  decoders, as security hole, 33
  encoding, 32–33, 33f
UTF-16, 40

V
Valgrind tool, 219–221, 512
Validation, 500. See also Input validation
Variable-length arrays (VLAs), 150–151
Variadic functions, 309–313, 344–346
vprintf() function, 340
Vector registers, 357–358
Vectored exception handling (VEH), 136–137
Vectorization, 358
VEH. See Vectored exception handling (VEH)
Venema, Wietse, 502–503
vprintf() function, 314
Viega, John, 498
Virtual function(s), 131–132
Virtual function table (VTBL), 132–133, 132f
Virtual pointer (VPTR), 132–133, 132f
Visibility, and thread safety, 370
Visual C++, 26
  /GS and function protection, 108–109
  /GS flag, 503–504, 504f
  security diagnostics, 507
  stack canary implementation, 108
Visual C++ 2012
  autovectorizer, 358–359
  loop pragma, 358
  /Qpar compiler switch, 358
Visual Studio
  C4244 warning, 305
  compiler-generated runtime checks, 106, 300
  /GS flag, 504–505
  /sd1 switch, 505
  stack buffer overrun detection, 108–109
Visual Studio 2010, formatted output vulnerability, 326f
VLAs. See Variable-length arrays (VLAs)
volatile type qualifier, 366–368
vprintf() function, 314
VPTR (virtual pointer), 132–133, 132f
vsnprintf() function, 314, 339–340
vprintf() function, 314
VTBL (virtual function table), 132–133, 132f
Vulnerability(ies), 21
  in ActiveX controls, 515
  buffer overflow, 117–118
  concurrency, 399–401
    DoS attacks in multicore DRAM systems, 399
  in system call wrappers, 400–401
    time-of-audit-to-time-of-use (TOATTOU), 401
  time-of-check-to-time-of-use (TOCTTOU), 401
  time-of-replacement-to-time-of-use (TORTTOU), 401
definition, 15
disclosure of, by hackers, 8–9
double-free, 157, 158, 160, 177–178. See also CV$ server double-free
dlmalloc, 191–195
RtlHeap, 208–211
dynamic memory management, 222–224
  in existing code, 495–496
file I/O
  directory traversal, 432–435
  path equivalence, 435–437
  privilege escalation, 418
  symlink-related, 438–439
  time of check, time of use (TOCTOU), 451–453, 455
filtering out, in software development, 479–480
format string. See Format string vulnerability(ies)
formatted output
  buffer overflow, 319–321
  CDE ToolTalk, 348–349
  crashing a program, 321–322
  direct argument access, 335–337
  Ettercap version NG-0.7.2, 349
  internationalization, 331
  output streams, 321
  overwriting memory, 326–331
  viewing memory content, 324–326, 325f
  viewing stack content, 322–324, 323f
  Washington University FTP daemon, 348
  wide-character, 332
  writing addresses in two words, 334–335
heap-based, 196–212
  mitigation strategies, 212–222
integer, 283–288. See also Integer wraparound
  conversion errors, 285
  nonexceptional integer logic errors, 287–288
  truncation errors, 285–287
  intentional, 16
in Microsoft Office versus OpenOffice, 474, 475f
in programs, versus in systems and networks, 16
  security flaws and, 15
  string, 50–72, 117–118
Vulnerability analyst, definition, 13
Vulnerability reports, sources of, 11

W
W xor X. See W^X policy
wa11 program, 422
Warren, Henry S., 299
Washington University FTP daemon, 348
Watson, Robert, 400
W32.Blaster.Worm, 1–5, 2f, 117
  flawed logic exploited by, 5, 5f
wchar_t, 30, 35, 39, 40
wcslen() function, 41
-Wformat flag, 343
-Wformat-nonliteral flag, 343
-Wformat-security flag, 343
Whitelisting, 111, 502–503
Wide string(s), 33–34
  sizing, 40–41
Wide-character input/output functions, 408, 412
Wide-character vulnerability, 332
Widening-multiplication instruction, 271
Win32
  CRT memory functions, 197–198, 197f
  heap memory API, 197, 197f
  local, global memory API, 197, 197f
  memory management APIs, 196, 197f
  memory-mapped file API, 197f, 198
  virtual memory API, 196–197, 197f
<table>
<thead>
<tr>
<th>Index</th>
<th>569</th>
</tr>
</thead>
</table>

**Windows**, 26  
address space layout randomization, 112–113  
process memory organization, 54, 55  
data declaration and, 123  

Wing, Jeannette, 517  

Worms, damage potential of, 4  

Wraparound, 229–231  

Wrappers, secure, 496  

Writing addresses in two words, 334–335  

Writing to freed memory  
dlmalloc, 195–196  
RtlHeap, 207–208  

WU-FTP, format string vulnerability, 319  

wu-ftpd vulnerability, 348  
W^X policy, 113–115, 140  

**X**  
XD (eXecute Disable) bit, 114  

Xfocus, 3  

XN (eXecute Never) bit, 114  

**Y**  
Yu, Fang, et al., symbolic string verification technique, 306  

**Z**  
zzuf tool, 514, 514n