THE ART OF
SOFTWARE SECURITY ASSESSMENT
This page intentionally left blank
# Table of Contents

ABOUT THE AUTHORS  xv
PREFACE  xvii
ACKNOWLEDGMENTS  xxi

1 Introduction to Software Security Assessment

1 SOFTWARE VULNERABILITY FUNDAMENTALS  3
   Introduction  3
   Vulnerabilities  4
   Security Policies  5
   Security Expectations  7
   The Necessity of Auditing  9
   Auditing Versus Black Box Testing  11
   Code Auditing and the Development Life Cycle  13
   Classifying Vulnerabilities  14
   Design Vulnerabilities  14
   Implementation Vulnerabilities  15
   Operational Vulnerabilities  16
   Gray Areas  17

Common Threads  18
   Input and Data Flow  18
   Trust Relationships  19
   Assumptions and Misplaced Trust  20
   Interfaces  21
   Environmental Attacks  21
   Exceptional Conditions  22
   Summary  23

2 DESIGN REVIEW  25
   Introduction  25
   Software Design Fundamentals  26
      Algorithms  26
      Abstraction and Decomposition  27
      Trust Relationships  28
      Principles of Software Design  31
      Fundamental Design Flaws  33
   Enforcing Security Policy  36
      Authentication  36
      Authorization  38
      Accountability  40
      Confidentiality  41
## Table of Contents

1. **Application Review Process** 91
   - Introduction 91
   - Overview of the Application Review Process 92
   - Rationale 92
   - Process Outline 93
   - Preassessment 93
   - Scoping 94
   - Application Access 95
   - Information Collection 96
   - Application Review 97
   - Avoid Drowning 98
   - Iterative Process 98
   - Initial Preparation 99
   - Plan 101
   - Work 103
   - Reflect 105
   - Documentation and Analysis 106
   - Reporting and Remediation Support 108
   - Code Navigation 109
     - External Flow Sensitivity 109
     - Tracing Direction 111
   - Code-Auditing Strategies 111
   - Code Comprehension Strategies 113
   - Candidate Point Strategies 119
   - Design Generalization Strategies 128
   - Code-Auditing Techniques 133
     - Internal Flow Analysis 133
     - Subsystem and Dependency Analysis 135
     - Rereading Code 136
     - Desk-Checking 137

2. **Operational Review** 67
   - Introduction 67
   - Exposure 68
     - Attack Surface 68
     - Insecure Defaults 69
     - Access Control 69
   - Unnecessary Services 70
   - Secure Channels 71
   - Spoofing and Identification 72
   - Network Profiles 73
   - Web-Specific Considerations 73
     - HTTP Request Methods 73
     - Directory Indexing 74
     - File Handlers 74
   - Authentication 75
   - Default Site Installations 75
   - Overly Verbose Error Messages 75
   - Public-Facing Administrative Interfaces 76
   - Protective Measures 76
   - Development Measures 76
   - Host-Based Measures 79
   - Network-Based Measures 83
   - Summary 89

3. **Integrity** 45
   - Availability 48

4. **Threat Modeling** 49
   - Information Collection 50
   - Application Architecture Modeling 53
   - Threat Identification 59
   - Documentation of Findings 62
   - Prioritizing the Implementation Review 65
   - Summary 66

5. **Application Architecture Modeling** 53

6. **Threat Identification** 59

7. **Documentation of Findings** 62

8. **Prioritizing the Implementation Review** 65

9. **Summary** 66
# Table of Contents

Usual Arithmetic Conversion

- Applications 242

Type Conversion Summary 244

Type Conversion

- Vulnerabilities 246

Signed/Unsigned Conversions 246

- Sign Extension 248

Truncation 259

Comparisons 265

Operators 271

- The sizeof Operator 271

Unexpected Results 272

Pointer Arithmetic 277

- Pointer Overview 277

- Pointer Arithmetic Overview 278

Vulnerabilities 280

Other C Nuances 282

- Order of Evaluation 282

- Structure Padding 284

- Precedence 287

- Macros/Preprocessor 288

- Typos 289

Summary 296

## 7 Program Building Blocks 297

Introduction 297

Auditing Variable Use 298

- Variable Relationships 298

- Structure and Object Mismanagement 307

- Variable Initialization 312

- Arithmetic Boundaries 316

- Type Confusion 319

- Lists and Tables 321

Auditing Control Flow 326

- Looping Constructs 327

Flow Transfer Statements 336

- Switch Statements 337

Auditing Functions 339

- Function Audit Logs 339

- Return Value Testing and Interpretation 340

- Function Side-Effects 351

- Argument Meaning 360

Auditing Memory

- Management 362

- ACC Logs 362

- Allocation Functions 369

- Allocator Scorecards and Error Domains 377

- Double-Frees 379

Summary 385

## 8 Strings and Metacharacters 387

Introduction 387

C String Handling 388

- Unbounded String Functions 388

- Bounded String Functions 393

- Common Issues 400

Metacharacters 407

- Embedded Delimiters 408

- NUL Character Injection 411

- Truncation 414

Common Metacharacter Formats 418

- Path Metacharacters 418

- C Format Strings 422

- Shell Metacharacters 425

- Perl open() 429

- SQL Queries 431

Metacharacter Filtering 434

- Eliminating Metacharacters 434
# Table of Contents

- Escaping Metacharacters 439
- Metacharacter Evasion 441
- Character Sets and Unicode 446
- Unicode 446
- Windows Unicode Functions 450
- Summary 457

9 UNIX I: PRIVILEGES AND FILES 459

- Introduction 459
- UNIX 101 460
  - Users and Groups 461
  - Files and Directories 462
  - Processes 464
- Privilege Model 464
  - Privileged Programs 466
  - User ID Functions 468
  - Group ID Functions 475
- Privilege Vulnerabilities 477
  - Reckless Use of Privileges 477
  - Dropping Privileges
    - Permanently 479
  - Dropping Privileges
    - Temporarily 486
  - Auditing Privilege-Management Code 488
  - Privilege Extensions 491
  - File Security 494
    - File IDs 494
    - File Permissions 495
    - Directory Permissions 498
    - Privilege Management with File Operations 499
    - File Creation 500
    - Directory Safety 503
    - Filenames and Paths 503
    - Dangerous Places 507
  - Interesting Files 508
    - File Internals 512
  - File Descriptors 512
    - Inodes 513
    - Directories 514
  - Links 515
    - Symbolic Links 515
    - Hard Links 522
  - Race Conditions 526
    - TOCTOU 527
  - The stat() Family of Functions 528
    - File Race Redux 532
    - Permission Races 533
    - Ownership Races 534
    - Directory Races 535
    - Temporary Files 538
    - Unique File Creation 538
    - File Reuse 544
    - Temporary Directory Cleaners 546
    - The Stdio File Interface 547
  - Opening a File 548
  - Reading from a File 550
  - Writing to a File 555
  - Closing a File 556
  - Summary 557

10 UNIX II: PROCESSES 559

- Introduction 559
- Processes 560
  - Process Creation 560
  - fork() Variants 562
  - Process Termination 562
  - fork() and Open Files 563
  - Program Invocation 565
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Windows I: Objects and the File System</td>
<td>625</td>
</tr>
<tr>
<td></td>
<td>Introduction</td>
<td>625</td>
</tr>
<tr>
<td></td>
<td>Background</td>
<td>626</td>
</tr>
<tr>
<td></td>
<td>Objects</td>
<td>627</td>
</tr>
<tr>
<td></td>
<td>Object Namespaces</td>
<td>629</td>
</tr>
<tr>
<td></td>
<td>Object Handles</td>
<td>632</td>
</tr>
<tr>
<td></td>
<td>Sessions</td>
<td>636</td>
</tr>
<tr>
<td></td>
<td>Security IDs</td>
<td>637</td>
</tr>
<tr>
<td></td>
<td>Logon Rights</td>
<td>638</td>
</tr>
<tr>
<td></td>
<td>Access Tokens</td>
<td>639</td>
</tr>
<tr>
<td></td>
<td>Security Descriptors</td>
<td>647</td>
</tr>
<tr>
<td></td>
<td>Access Masks</td>
<td>648</td>
</tr>
<tr>
<td></td>
<td>ACL Inheritance</td>
<td>649</td>
</tr>
<tr>
<td></td>
<td>Security Descriptors Programming Interfaces</td>
<td>649</td>
</tr>
<tr>
<td></td>
<td>Auditing ACL Permissions</td>
<td>652</td>
</tr>
<tr>
<td></td>
<td>Processes and Threads</td>
<td>654</td>
</tr>
<tr>
<td></td>
<td>Process Loading</td>
<td>654</td>
</tr>
<tr>
<td></td>
<td>ShellExecute and ShellExecuteEx</td>
<td>655</td>
</tr>
<tr>
<td></td>
<td>DLL Loading</td>
<td>656</td>
</tr>
<tr>
<td></td>
<td>Services</td>
<td>658</td>
</tr>
<tr>
<td></td>
<td>File Access</td>
<td>659</td>
</tr>
<tr>
<td></td>
<td>The File I/O API</td>
<td>661</td>
</tr>
<tr>
<td></td>
<td>Links</td>
<td>676</td>
</tr>
<tr>
<td></td>
<td>The Registry</td>
<td>680</td>
</tr>
<tr>
<td></td>
<td>Key Permissions</td>
<td>681</td>
</tr>
<tr>
<td></td>
<td>Key and Value Squatting</td>
<td>682</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>684</td>
</tr>
<tr>
<td>12</td>
<td>Windows II: Interprocess Communication</td>
<td>685</td>
</tr>
<tr>
<td></td>
<td>Introduction</td>
<td>685</td>
</tr>
<tr>
<td></td>
<td>Windows IPC Security</td>
<td>686</td>
</tr>
<tr>
<td></td>
<td>The Redirector</td>
<td>686</td>
</tr>
<tr>
<td></td>
<td>Impersonation</td>
<td>688</td>
</tr>
<tr>
<td></td>
<td>Window Messaging</td>
<td>689</td>
</tr>
<tr>
<td></td>
<td>Window Stations Object</td>
<td>690</td>
</tr>
<tr>
<td></td>
<td>The Desktop Object</td>
<td>690</td>
</tr>
<tr>
<td></td>
<td>Window Messages</td>
<td>691</td>
</tr>
<tr>
<td></td>
<td>Shatter Attacks</td>
<td>694</td>
</tr>
<tr>
<td></td>
<td>DDE</td>
<td>697</td>
</tr>
<tr>
<td></td>
<td>Terminal Sessions</td>
<td>697</td>
</tr>
<tr>
<td></td>
<td>Pipes</td>
<td>698</td>
</tr>
<tr>
<td></td>
<td>Pipe Permissions</td>
<td>698</td>
</tr>
<tr>
<td></td>
<td>Named Pipes</td>
<td>699</td>
</tr>
<tr>
<td></td>
<td>Pipe Creation</td>
<td>699</td>
</tr>
<tr>
<td></td>
<td>Impersonation in Pipes</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>Pipe Squatting</td>
<td>703</td>
</tr>
</tbody>
</table>
Table of Contents

Mailslots 705
Mailslot Permissions 705
Mailslot Squatting 706
Remote Procedure Calls 706
RPC Connections 706
RPC Transports 707
Microsoft Interface Definition Language 708
IDL File Structure 708
Application Configuration Files 710
RPC Servers 711
Impersonation in RPC 716
Context Handles and State 718
Threading in RPC 721
Auditing RPC Applications 722
COM 725
COM: A Quick Primer 725
DCOM Configuration Utility 731
DCOM Application Identity 732
DCOM Subsystem Access Permissions 733
DCOM Access Controls 734
Impersonation in DCOM 736
MIDL Revisited 738
Active Template Library 740
Auditing DCOM Applications 741
ActiveX Security 749
Summary 754

13 SYNCHRONIZATION AND STATE 755
Introduction 755
Synchronization Problems 756
Reentrancy and Asynchronous-Safe Code 757
Race Conditions 759
Starvation and Deadlocks 760
Process Synchronization 762
System V Process Synchronization 762
Windows Process Synchronization 765
Vulnerabilities with Interprocess Synchronization 770
Signals 783
Sending Signals 786
Handling Signals 786
Jump Locations 788
Signal Vulnerabilities 791
Signals Scoreboard 809
Threads 810
PThreads API 811
Windows API 813
Threading Vulnerabilities 815
Summary 825

III SOFTWARE VULNERABILITIES IN PRACTICE

14 NETWORK PROTOCOLS 829
Introduction 829
Internet Protocol 831
IP Addressing Primer 832
IP Packet Structures 834
Basic IP Header Validation 836
IP Options Processing 844
Source Routing 851
Fragmentation 853
User Datagram Protocol 863
Basic UDP Header Validation 864
UDP Issues 864
Transmission Control Protocol 864
Basic TCP Header Validation 866
TCP Options Processing 867
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNS Names</td>
<td>993</td>
</tr>
<tr>
<td>Length Variables</td>
<td>996</td>
</tr>
<tr>
<td>DNS Spoofing</td>
<td>1002</td>
</tr>
<tr>
<td>Summary</td>
<td>1005</td>
</tr>
<tr>
<td><strong>17 WEB APPLICATIONS 1007</strong></td>
<td></td>
</tr>
<tr>
<td>Introduction</td>
<td>1007</td>
</tr>
<tr>
<td>Web Technology Overview</td>
<td>1008</td>
</tr>
<tr>
<td>The Basics</td>
<td>1009</td>
</tr>
<tr>
<td>Static Content</td>
<td>1009</td>
</tr>
<tr>
<td>CGI</td>
<td>1009</td>
</tr>
<tr>
<td>Web Server APIs</td>
<td>1010</td>
</tr>
<tr>
<td>Server-Side Includes</td>
<td>1011</td>
</tr>
<tr>
<td>Server-Side Transformation</td>
<td>1012</td>
</tr>
<tr>
<td>Server-Side Scripting</td>
<td>1013</td>
</tr>
<tr>
<td>HTTP</td>
<td>1014</td>
</tr>
<tr>
<td>Overview</td>
<td>1014</td>
</tr>
<tr>
<td>Versions</td>
<td>1017</td>
</tr>
<tr>
<td>Headers</td>
<td>1018</td>
</tr>
<tr>
<td>Methods</td>
<td>1020</td>
</tr>
<tr>
<td>Parameters and Forms</td>
<td>1022</td>
</tr>
<tr>
<td>State and HTTP</td>
<td></td>
</tr>
<tr>
<td>Authentication</td>
<td>1027</td>
</tr>
<tr>
<td>Overview</td>
<td>1028</td>
</tr>
<tr>
<td>Client IP Addresses</td>
<td>1029</td>
</tr>
<tr>
<td>Referer Request Header</td>
<td>1030</td>
</tr>
<tr>
<td>Embedding State in HTML and URLs</td>
<td>1032</td>
</tr>
<tr>
<td>HTTP Authentication</td>
<td>1033</td>
</tr>
<tr>
<td>Cookies</td>
<td>1036</td>
</tr>
<tr>
<td>Sessions</td>
<td>1038</td>
</tr>
<tr>
<td>Architecture</td>
<td>1040</td>
</tr>
<tr>
<td>Redundancy</td>
<td>1040</td>
</tr>
<tr>
<td>Presentation Logic</td>
<td>1040</td>
</tr>
<tr>
<td>Business Logic</td>
<td>1041</td>
</tr>
<tr>
<td>N-Tier Architectures</td>
<td>1041</td>
</tr>
<tr>
<td>Business Tier</td>
<td>1043</td>
</tr>
<tr>
<td>Web Tier:</td>
<td></td>
</tr>
<tr>
<td>Model-View-Controller</td>
<td>1044</td>
</tr>
<tr>
<td>Problem Areas</td>
<td>1046</td>
</tr>
<tr>
<td>Client Visibility</td>
<td>1046</td>
</tr>
<tr>
<td>Client Control</td>
<td>1047</td>
</tr>
<tr>
<td>Page Flow</td>
<td>1048</td>
</tr>
<tr>
<td>Sessions</td>
<td>1049</td>
</tr>
<tr>
<td>Authentication and Access Control</td>
<td>1056</td>
</tr>
<tr>
<td>Authentication</td>
<td>1056</td>
</tr>
<tr>
<td>Authorization and Access Control</td>
<td>1057</td>
</tr>
<tr>
<td>Encryption and SSL/TLS</td>
<td>1058</td>
</tr>
<tr>
<td>Phishing and Impersonation</td>
<td>1059</td>
</tr>
<tr>
<td>Common Vulnerabilities</td>
<td>1060</td>
</tr>
<tr>
<td>SQL Injection</td>
<td>1061</td>
</tr>
<tr>
<td>OS and File System Interaction</td>
<td>1066</td>
</tr>
<tr>
<td>XML Injection</td>
<td>1069</td>
</tr>
<tr>
<td>XPath Injection</td>
<td>1070</td>
</tr>
<tr>
<td>Cross-Site Scripting</td>
<td>1071</td>
</tr>
<tr>
<td>Threading Issues</td>
<td>1074</td>
</tr>
<tr>
<td>C/C++ Problems</td>
<td>1075</td>
</tr>
<tr>
<td>Harsh Realities of the Web</td>
<td>1075</td>
</tr>
<tr>
<td>Auditing Strategy</td>
<td>1078</td>
</tr>
<tr>
<td>Summary</td>
<td>1081</td>
</tr>
<tr>
<td><strong>18 WEB TECHNOLOGIES 1083</strong></td>
<td></td>
</tr>
<tr>
<td>Introduction</td>
<td>1083</td>
</tr>
<tr>
<td>Web Services and Service-Oriented Architecture</td>
<td>1084</td>
</tr>
<tr>
<td>SOAP</td>
<td>1085</td>
</tr>
<tr>
<td>REST</td>
<td>1085</td>
</tr>
<tr>
<td>AJAX</td>
<td>1085</td>
</tr>
<tr>
<td>Web Application Platforms</td>
<td>1086</td>
</tr>
<tr>
<td>CGI</td>
<td>1086</td>
</tr>
<tr>
<td>Indexed Queries</td>
<td>1086</td>
</tr>
<tr>
<td>Environment Variables</td>
<td>1087</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Path Confusion</td>
<td>1091</td>
</tr>
<tr>
<td>Perl</td>
<td>1093</td>
</tr>
<tr>
<td>SQL Injection</td>
<td>1093</td>
</tr>
<tr>
<td>File Access</td>
<td>1094</td>
</tr>
<tr>
<td>Shell Invocation</td>
<td>1095</td>
</tr>
<tr>
<td>File Inclusion</td>
<td>1095</td>
</tr>
<tr>
<td>Inline Evaluation</td>
<td>1095</td>
</tr>
<tr>
<td>Cross-Site Scripting</td>
<td>1096</td>
</tr>
<tr>
<td>Taint Mode</td>
<td>1096</td>
</tr>
<tr>
<td>PHP</td>
<td>1096</td>
</tr>
<tr>
<td>SQL Injection</td>
<td>1097</td>
</tr>
<tr>
<td>File Access</td>
<td>1098</td>
</tr>
<tr>
<td>Shell Invocation</td>
<td>1099</td>
</tr>
<tr>
<td>File Inclusion</td>
<td>1101</td>
</tr>
<tr>
<td>Inline Evaluation</td>
<td>1101</td>
</tr>
<tr>
<td>Cross-Site Scripting</td>
<td>1103</td>
</tr>
<tr>
<td>Configuration</td>
<td>1104</td>
</tr>
<tr>
<td>Java</td>
<td>1105</td>
</tr>
<tr>
<td>SQL Injection</td>
<td>1106</td>
</tr>
<tr>
<td>File Access</td>
<td>1107</td>
</tr>
<tr>
<td>Shell Invocation</td>
<td>1108</td>
</tr>
<tr>
<td>File Inclusion</td>
<td>1108</td>
</tr>
<tr>
<td>JSP File Inclusion</td>
<td>1109</td>
</tr>
<tr>
<td>Inline Evaluation</td>
<td>1110</td>
</tr>
<tr>
<td>Cross-Site Scripting</td>
<td>1110</td>
</tr>
<tr>
<td>Threading Issues</td>
<td>1111</td>
</tr>
<tr>
<td>Configuration</td>
<td>1112</td>
</tr>
<tr>
<td>ASP</td>
<td>1113</td>
</tr>
<tr>
<td>SQL Injection</td>
<td>1113</td>
</tr>
<tr>
<td>File Access</td>
<td>1115</td>
</tr>
<tr>
<td>Shell Invocation</td>
<td>1115</td>
</tr>
<tr>
<td>File Inclusion</td>
<td>1116</td>
</tr>
<tr>
<td>Inline Evaluation</td>
<td>1117</td>
</tr>
<tr>
<td>Cross-Site Scripting</td>
<td>1118</td>
</tr>
<tr>
<td>Configuration</td>
<td>1118</td>
</tr>
<tr>
<td>ASP.NET</td>
<td>1118</td>
</tr>
<tr>
<td>SQL Injection</td>
<td>1118</td>
</tr>
<tr>
<td>File Access</td>
<td>1119</td>
</tr>
<tr>
<td>Shell Invocation</td>
<td>1120</td>
</tr>
<tr>
<td>File Inclusion</td>
<td>1120</td>
</tr>
<tr>
<td>Inline Evaluation</td>
<td>1121</td>
</tr>
<tr>
<td>Cross-Site Scripting</td>
<td>1121</td>
</tr>
<tr>
<td>Configuration</td>
<td>1121</td>
</tr>
<tr>
<td>ViewState</td>
<td>1121</td>
</tr>
<tr>
<td>Summary</td>
<td>1123</td>
</tr>
</tbody>
</table>

**BIBLIOGRAPHY** 1125

**INDEX** 1129
About the Authors

Mark Dowd is a principal security architect at McAfee, Inc. and an established expert in the field of application security. His professional experience includes several years as a senior researcher at Internet Security Systems (ISS) X-Force, and the discovery of a number of high-profile vulnerabilities in ubiquitous Internet software. He is responsible for identifying and helping to address critical flaws in Sendmail, Microsoft Exchange Server, OpenSSH, Internet Explorer, Mozilla (Firefox), Checkpoint VPN, and Microsoft’s SSL implementation. In addition to his research work, Mark presents at industry conferences, including Black Hat and RUXCON.

John McDonald is a senior consultant with Neohapsis, where he specializes in advanced application security assessment across a broad range of technologies and platforms. He has an established reputation in software security, including work in security architecture and vulnerability research for NAI (now McAfee), Data Protect GmbH, and Citibank. As a vulnerability researcher, John has identified and helped resolve numerous critical vulnerabilities, including issues in Solaris, BSD, Checkpoint FireWall-1, OpenSSL, and BIND.

Justin Schuh is a senior consultant with Neohapsis, where he leads the Application Security Practice. As a senior consultant and practice lead, he performs software security assessments across a range of systems, from embedded device firmware to distributed enterprise web applications. Prior to his employment with Neohapsis, Justin spent nearly a decade in computer security activities at the Department of Defense (DoD) and related agencies. His government service includes a role as a lead researcher with the National Security Agency (NSA) penetration testing team—the Red Team.
This page intentionally left blank
Preface

“If popular culture has taught us anything, it is that someday mankind must face and destroy the growing robot menace.”

Daniel H. Wilson, How to Survive a Robot Uprising

The past several years have seen huge strides in computer security, particularly in the field of software vulnerabilities. It seems as though every stop at the bookstore introduces a new title on topics such as secure development or exploiting software. Books that cover application security tend to do so from the perspective of software designers and developers and focus on techniques to prevent software vulnerabilities from occurring in applications. These techniques start with solid security design principles and threat modeling and carry all the way through to implementation best practices and defensive programming strategies. Although they serve as strong defensive foundations for application development, these resources tend to give little treatment to the nature of vulnerabilities; instead, they focus on how to avoid them. What’s more, every development team can’t start rebuilding a secure application from the ground up. Real people have to deal with huge existing codebases, in-place applications, and limited time and budget. Meanwhile, the secure coding mantra seems to be “If it smells bad, throw it out.” That’s certainly necessary in some cases, but often it’s too expensive and time consuming to be reasonable. So you might turn your attention to penetration testing and ethical hacking instead. A wide range of information on this topic is available, and it’s certainly useful for the acid test of a software system. However, even the most technically detailed resources have a strong focus on exploit development and little to no treatment on how to find vulnerabilities in the first place. This still leaves the hanging question of how to find issues in an existing application and how to get a reasonable degree of assurance that a piece of software is safe.
This problem is exactly the one faced by those in the field of professional software security assessment. People are growing more concerned with building and testing secure systems, but very few resources address the practice of finding vulnerabilities. After all, this process requires a deep technical understanding of some very complex issues and must include a systematic approach to analyzing an application. Without formally addressing how to find vulnerabilities, the software security industry has no way of establishing the quality of a software security assessment or training the next generation in the craft. We have written this book in the hope of answering these questions and to help bridge the gap between secure software development and practical post-implementation reviews. Although this book is aimed primarily at consultants and other security professionals, much of the material will have value to the rest of the IT community as well. Developers can gain insight into the subtleties and nuances of how languages and operating systems work and how those features can introduce vulnerabilities into an application that otherwise appears secure. Quality assurance (QA) personnel can use some of the guidelines in this book to ensure the integrity of in-house software and cut down on the likelihood of their applications being stung by a major vulnerability. Administrators can find helpful guidelines for evaluating the security impact of applications on their networks and use this knowledge to make better decisions about future deployments. Finally, hobbyists who are simply interested in learning more about how to assess applications will find this book an invaluable resource (we hope!) for getting started in application security review or advancing their current skill sets.

Prerequisites
The majority of this book has been targeted at a level that any moderately experienced developer should find approachable. This means you need to be fairly comfortable with at least one programming language, and ideally, you should be familiar with basic C/C++ programming. At several stages throughout the book, we use Intel assembly examples, but we have attempted to keep them to a minimum and translate them into approximate C code when possible. We have also put a lot of effort into making the material as platform neutral as possible, although we do cover platform specifics for the most common operating systems. When necessary, we have tried to include references to additional resources that provide background for material that can’t be covered adequately in this book.

How to Use This Book
Before we discuss the use of this book, we need to introduce its basic structure. The book is divided into three different parts:
• *Part I: Introduction to Software Security Assessment (Chapters 1–4)*—These chapters introduce the practice of code auditing and explain how it fits into the software development process. You learn about the function of design review, threat modeling, and operational review—tools that are useful for evaluating an application as a whole, and not just the code. Finally, you learn some generic high-level methods for performing a code review on any application, regardless of its function or size.

• *Part II: Software Vulnerabilities (Chapters 5–13)*—These chapters shift the focus of the book toward practical implementation review and address how to find specific vulnerabilities in an application’s codebase. Major software vulnerability classes are described, and you learn how to discover high-risk security flaws in an application. Numerous real-world examples of security vulnerabilities are given to help you get a feel for what software bugs look like in real code.

• *Part III: Software Vulnerabilities in Practice (Chapters 14–18)*—The final portion of the book turns your attention toward practical uses of lessons learned from the earlier chapters. These chapters describe a number of common application classes and the types of bugs they tend to be vulnerable to. They also show you how to apply the technical knowledge gained from Part II to real-world applications. Specifically, you look at networking, firewalling technologies, and Web technologies. Each chapter in this section introduces the common frameworks and designs of each application class and identifies where flaws typically occur.

You’ll get the most value if you read this book straight through at least once so that you can get a feel for the material. This approach is best because we have tried to use each section as an opportunity to highlight techniques and tools that help you in performing application assessments. In particular, you should pay attention to the sidebars and notes we use to sum up the more important concepts in a section.

Of course, busy schedules and impending deadlines can have a serious impact on your time. To that end, we want to lay out a few tracks of focus for different types of reviews. However, you should start with Part 1 (Chapters 1–4) because it establishes a foundation for the rest of the book. After that, you can branch out to the following chapters:

• *UNIX track (Chapters 5–10, 13)*—This chapter track starts off by covering common software vulnerability classes, such as memory corruption, program control flow, and specially formatted data. Then UNIX-centered security problems that arise because of quirks in the various UNIX operating systems are addressed. Finally, this track ends with coverage of synchronization vulnerabilities common to most platforms.
• **Windows track (Chapters 5–8, 11–13)—**This track starts off similarly to the UNIX track, by covering platform-neutral security problems. Then two chapters specifically address Windows APIs and their related vulnerabilities. Finally, this track finishes with coverage of common synchronization vulnerabilities.

• **Web track (Chapters 8, 13, 17, 18)—**Web auditing requires understanding common security vulnerabilities as well as Web-based frameworks and languages. This track discusses the common vulnerability classes that pertain to Web-based languages, and then finishes off with the Web-specific chapters. Although the UNIX and Windows chapters aren’t listed here, reading them might be necessary depending on the Web application’s deployment environment.

• **Network application track (Chapters 5–8, 13, 16)—**This sequence of chapters best addresses the types of vulnerabilities you’re likely to encounter with network client/server applications. Notice that even though Chapter 16 is targeted at selected application protocols, it has a section for generic application protocol auditing methods. Like the previous track, UNIX or Windows chapters might also be relevant, depending on the deployment environment.

• **Network analysis track (Chapters 5–8, 13–16)—**This track is aimed at analyzing network analysis applications, such as firewalls, IPSs, sniffers, routing software, and so on. Coverage includes standard vulnerability classes along with popular network-based technologies and the common vulnerabilities in these products. Again, the UNIX and Windows chapters would be a good addition to this track, if applicable.
Acknowledgments

Mark: To my family, friends, and colleagues, for supporting me and providing encouragement throughout this endeavor.

John: To my girlfriend Jess, my family and friends, Neohapsis, Vincent Howard, Dave Aitel, David Leblanc, Thomas Lopatic, and Howard Kirk.

Justin: To my wife Cat, my coworkers at Neohapsis, my family and friends, and everyone at a three-letter agency who kept me out of trouble.

We would collectively like to thank reviewers, friends, and colleagues who have given invaluable feedback, suggestions, and comments that helped shape this book into the finished product you see today. In particular, we would like to acknowledge Neel Mehta, Halvar Flake, John Viega, and Nishad Herath for their tireless efforts in reviewing and helping to give us technical and organizational direction. We’d also like to thank the entire publishing team at Addison-Wesley for working with us to ensure the highest-quality finished product possible.
This page intentionally left blank
Chapter 6

C Language Issues

“One day you will understand.”
Neel Mehta, Senior Researcher, Internet Security Systems X-Force

Introduction

When you’re reviewing software to uncover potential security holes, it’s important to understand the underlying details of how the programming language implements data types and operations, and how those details can affect execution flow. A code reviewer examining an application binary at the assembly level can see explicitly how data is stored and manipulated as well as the exact implications of an operation on a piece of data. However, when you’re reviewing an application at the source code level, some details are abstracted and less obvious. This abstraction can lead to the introduction of subtle vulnerabilities in software that remain unnoticed and uncorrected for long periods of time. A thorough auditor should be familiar with the source language’s underlying implementation and how these details can lead to security-relevant conditions in border cases or exceptional situations.
This chapter explores subtle details of the C programming language that could adversely affect an application’s security and robustness. Specifically, it covers the storage details of primitive types, arithmetic overflow and underflow conditions, type conversion issues, such as the default type promotions, signed/unsigned conversions and comparisons, sign extension, and truncation. You also look at some interesting nuances of C involving unexpected results from certain operators and other commonly unappreciated behaviors. Although this chapter focuses on C, many principles can be applied to other languages.

C Language Background

This chapter deals extensively with specifics of the C language and uses terminology from the C standards. You shouldn’t have to reference the standards to follow this material, but this chapter makes extensive use of the public final draft of the C99 standard (ISO/IEC 9899:1999), which you can find at www.open-std.org/jtc1/sc22/wg14/www/standards.

The C Rationale document that accompanies the draft standard is also useful. Interested readers should check out Peter Van der Linden’s excellent book Expert C Programming (Prentice Hall, 1994) and the second edition of Kernighan and Ritchie’s The C Programming Language (Prentice Hall, 1988). You might also be interested in purchasing the final version of the ISO standard or the older ANSI standard; both are sold through the ANSI organization’s Web site (www.ansi.org).

Although this chapter incorporates a recent standard, the content is targeted toward the current mainstream use of C, specifically the ANSI C89/ISO 90 standards. Because low-level security details are being discussed, notes on any situations in which changes across versions of C are relevant have been added.

Occasionally, the terms “undefined behavior” and “implementation-defined behavior” are used when discussing the standards. Undefined behavior is erroneous behavior: conditions that aren’t required to be handled by the compiler and, therefore, have unspecified results. Implementation-defined behavior is behavior that’s up to the underlying implementation. It should be handled in a consistent and logical manner, and the method for handling it should be documented.

Data Storage Overview

Before you delve into C’s subtleties, you should review the basics of C types—specifically, their storage sizes, value ranges, and representations. This section explains the types from a general perspective, explores details such as binary encoding, twos complement arithmetic, and byte order conventions, and winds up with some pragmatic observations on common and future implementations.
The C standards define an object as a region of data storage in the execution environment; its contents can represent values. Each object has an associated type: a way to interpret and give meaning to the value stored in that object. Dozens of types are defined in the C standards, but this chapter focuses on the following:

- **Character types**—There are three character types: `char`, `signed char`, and `unsigned char`. All three types are guaranteed to take up 1 byte of storage. Whether the char type is signed is implementation defined. Most current systems default to char being signed, although compiler flags are usually available to change this behavior.

- **Integer types**—There are four standard signed integer types, excluding the character types: `short int`, `int`, `long int`, and `long long int`. Each standard type has a corresponding unsigned type that takes the same amount of storage. (Note: The long long int type is new to C99.)

- **Floating types**—There are three real floating types and three complex types. The real floating types are `float`, `double`, and `long double`. The three complex types are `float _Complex`, `double _Complex`, and `long double _Complex`. (Note: The complex types are new to C99.)

- **Bit fields**—A bit field is a specific number of bits in an object. Bit fields can be signed or unsigned, depending on their declaration. If no sign type specifier is given, the sign of the bit field is implementation dependent.

**Note**

Bit fields might be unfamiliar to some programmers, as they usually aren’t present outside network code or low-level code. Here’s a brief example of a bit field:

```c
struct controller
{
    unsigned int id:4;
    unsigned int tflag:1;
    unsigned int rflag:1;
    unsigned int ack:2;
    unsigned int seqnum:8;
    unsigned int code:16;
};
```

The controller structure has several small members. id refers to a 4-bit unsigned variable, and tflag and rflag refer to single bits. ack is a 2-bit variable, seqnum is an 8-bit variable, and code is a 16-bit variable.
The members of this structure are likely to be laid out so that they’re contiguous bits in memory that fit within one 32-bit region.

From an abstract perspective, each integer type (including character types) represents a different integer size that the compiler can map to an appropriate underlying architecture-dependent data type. A character is guaranteed to consume 1 byte of storage (although a byte might not necessarily be 8 bits). `sizeof(char)` is always one, and you can always use an unsigned character pointer, `sizeof`, and `memcpy()` to examine and manipulate the actual contents of other types. The other integer types have certain ranges of values they are required to be able to represent, and they must maintain certain relationships with each other (long can’t be smaller than short, for example), but otherwise, their implementation largely depends on their architecture and compiler.

Signed integer types can represent both positive and negative values, whereas unsigned types can represent only positive values. Each signed integer type has a corresponding unsigned integer type that takes up the same amount of storage. Unsigned integer types have two possible types of bits: value bits, which contain the actual base-two representation of the object’s value, and padding bits, which are optional and otherwise unspecified by the standard. Signed integer types have value bits and padding bits as well as one additional bit: the sign bit. If the sign bit is clear in a signed integer type, its representation for a value is identical to that value’s representation in the corresponding unsigned integer type. In other words, the underlying bit pattern for the positive value 42 should look the same whether it’s stored in an int or unsigned int.

An integer type has a precision and a width. The precision is the number of value bits the integer type uses. The width is the number of bits the type uses to represent its value, including the value and sign bits, but not the padding bits. For unsigned integer types, the precision and width are the same. For signed integer types, the width is one greater than the precision.

Programmers can invoke the various types in several ways. For a given integer type, such as short int, a programmer can generally omit the int keyword. So the keywords signed short int, signed short, short int, and short refer to the same data type. In general, if the signed and unsigned type specifiers are omitted, the type is assumed to be signed. However, this assumption isn’t true for the char type, as whether it’s signed depends on the implementation. (Usually, chars are signed. If you need a signed character with 100% certainty, you can specifically declare a signed char.)

C also has a rich type-aliasing system supported via typedef, so programmers usually have preferred conventions for specifying a variable of a known size and representation. For example, types such as int8_t, uint8_t, int32_t, and u_int32_t are popular with UNIX and network programmers. They represent an 8-bit signed
integer, an 8-bit unsigned integer, a 32-bit signed integer, and a 32-bit unsigned integer, respectively. Windows programmers tend to use types such as BYTE, CHAR, and DWORD, which respectively map to an 8-bit unsigned integer, an 8-bit signed integer, and a 32-bit unsigned integer.

Binary Encoding

Unsigned integer values are encoded in pure binary form, which is a base-two numbering system. Each bit is a 1 or 0, indicating whether the power of two that the bit’s position represents is contributing to the number’s total value. To convert a positive number from binary notation to decimal, the value of each bit position $n$ is multiplied by $2^{n-1}$. A few examples of these conversions are shown in the following lines:

- $0001\ 1011 = 2^4 + 2^3 + 2^1 + 2^0 = 27$
- $0000\ 1111 = 2^3 + 2^2 + 2^1 + 2^0 = 15$
- $0010\ 1010 = 2^5 + 2^3 + 2^1 = 42$

Similarly, to convert a positive decimal integer to binary, you repeatedly subtract powers of two, starting from the highest power of two that can be subtracted from the integer leaving a positive result (or zero). The following lines show a few sample conversions:

- $55 = 32 + 16 + 4 + 2 + 1$
  $= (2^5) + (2^4) + (2^2) + (2^1) + (2^0)$
  $= 0011\ 0111$

- $37 = 32 + 4 + 1$
  $= (2^5) + (2^2) + (2^0)$
  $= 0010\ 0101$

Signed integers make use of a sign bit as well as value and padding bits. The C standards give three possible arithmetic schemes for integers and, therefore, three possible interpretations for the sign bit:

- **Sign and magnitude**—The sign of the number is stored in the sign bit. It’s 1 if the number is negative and 0 if the number is positive. The magnitude of the number is stored in the value bits. This scheme is easy for humans to read and understand but is cumbersome for computers because they have to explicitly compare magnitudes and signs for arithmetic operations.
Ones complement—Again, the sign bit is 1 if the number is negative and 0 if the number is positive. Positive values can be read directly from the value bits. However, negative values can’t be read directly; the whole number must be negated first. In ones complement, a number is negated by inverting all its bits. To find the value of a negative number, you have to invert its bits. This system works better for the machine, but there are still complications with addition, and, like sign and magnitude, it has the amusing ambiguity of having two values of zero: positive zero and negative zero.

Twos complement—The sign bit is 1 if the number is negative and 0 if the number is positive. You can read positive values directly from the value bits, but you can’t read negative values directly; you have to negate the whole number first. In twos complement, a number is negated by inverting all the bits and then adding one. This works well for the machine and removes the ambiguity of having two potential values of zero.

Integers are usually represented internally by using twos complement, especially in modern computers. As mentioned, twos complement encodes positive values in standard binary encoding. The range of positive values that can be represented is based on the number of value bits. A twos complement 8-bit signed integer has 7 value bits and 1 sign bit. It can represent the positive values 0 to 127 in the 7 value bits. All negative values represented with twos complement encoding require the sign bit to be set. The values from -128 to -1 can be represented in the value bits when the sign bit is set, thus allowing the 8-bit signed integer to represent -128 to 127.

For arithmetic, the sign bit is placed in the most significant bit of the data type. In general, a signed twos complement number of width X can represent the range of integers from $-2^{X-1}$ to $2^{X-1}-1$. Table 6-1 shows the typical ranges of twos complement integers of varying sizes.

Table 6-1

<table>
<thead>
<tr>
<th></th>
<th>8-bit</th>
<th>16-bit</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum value (signed)</td>
<td>-128</td>
<td>-32768</td>
<td>-2147483648</td>
<td>-9223372036854775808</td>
</tr>
<tr>
<td>Maximum value (signed)</td>
<td>127</td>
<td>32767</td>
<td>2147483647</td>
<td>9223372036854775807</td>
</tr>
<tr>
<td>Minimum value (unsigned)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Maximum value (unsigned)</td>
<td>255</td>
<td>65535</td>
<td>4294967295</td>
<td>18446744073709551615</td>
</tr>
</tbody>
</table>

As described previously, you negate a twos complement number by inverting all the bits and adding one. Listing 6-1 shows how you obtain the representation of -15 by inverting the number 15, and then how you figure out the value of an unknown negative bit pattern.
Listing 6-1

Twos Complement Representation of -15

0000 1111 – binary representation for 15
1111 0000 – invert all the bits
0000 0001 – add one
1111 0001 – twos complement representation for -15

1101 0110 – unknown negative number
0010 1001 – invert all the bits
0000 0001 – add one
0010 1010 – twos complement representation for 42
original number was -42

Byte Order

There are two conventions for ordering bytes in modern architectures: **big endian** and **little endian**. These conventions apply to data types larger than 1 byte, such as a short int or an int. In the big-endian architecture, the bytes are located in memory starting with the most significant byte and ending with the least significant byte. Little-endian architectures, however, start with the least significant byte and end with the most significant. For example, you have a 4-byte integer with the decimal value 12345. In binary, it’s 11000000111001. This integer is located at address 500. On a big-endian machine, it’s represented in memory as the following:

Address 500: 00000000
Address 501: 00000000
Address 502: 00110000
Address 503: 00111001

On a little-endian machine, however, it’s represented this way:

Address 500: 00111001
Address 501: 00110000
Address 502: 00000000
Address 503: 00000000

Intel machines are little endian, but RISC machines, such as SPARC, tend to be big endian. Some machines are capable of dealing with both encodings natively.

Common Implementations

Practically speaking, if you’re talking about a modern, 32-bit, twos complement machine, what can you say about C’s basic types and their representations?
In general, none of the integer types have any padding bits, so you don’t need to worry about that. Everything is going to use two's complement representation. Bytes are going to be 8 bits long. Byte order varies; it’s little endian on Intel machines but more likely to be big endian on RISC machines.

The char type is likely to be signed by default and take up 1 byte. The short type takes 2 bytes, and int takes 4 bytes. The long type is also 4 bytes, and long long is 8 bytes. Because you know integers are two's complement encoded and you know their underlying sizes, determining their minimum and maximum values is easy. Table 6-2 summarizes the typical sizes for ranges of integer data types on a 32-bit machine.

Table 6-2

<table>
<thead>
<tr>
<th>Type</th>
<th>Width (in Bits)</th>
<th>Minimum Value</th>
<th>Maximum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed char</td>
<td>8</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>-32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>Int</td>
<td>32</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>32</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>long long</td>
<td>64</td>
<td>-9,223,372,036,854,775,808</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>64</td>
<td>0</td>
<td>18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>

What can you expect in the near future as 64-bit systems become more prevalent? The following list describes a few type systems that are in use today or have been proposed:

- **ILP32**—int, long, and pointer are all 32 bits, the current standard for most 32-bit computers.
- **ILP32LL**—int, long, and pointer are all 32 bits, and a new type—long long—is 64 bits. The long long type is new to C99. It gives C a type that has a minimum width of 64 bits but doesn’t change any of the language’s fundamentals.
- **LP64**—long and pointer are 64 bits, so the pointer and long types have changed from 32-bit to 64-bit values.
- **ILP64**—int, long, and pointer are all 64 bits. The int type has been changed to a 64-bit type, which has fairly significant implications for the language.
- **LLP64**—pointers and the new long long type are 64 bits. The int and long types remain 32-bit data types.
Table 6-3 summarizes these type systems briefly.

Table 6-3

<table>
<thead>
<tr>
<th>Type</th>
<th>ILP32</th>
<th>ILP32LL</th>
<th>LP64</th>
<th>ILP64</th>
<th>LLP64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>long long</td>
<td>N/A</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>pointer</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>

As you can see, the typical data type sizes match the ILP32LL model, which is what most compilers adhere to on 32-bit platforms. The LP64 model is the de facto standard for compilers that generate code for 64-bit platforms. As you learn later in this chapter, the int type is a basic unit for the C language; many things are converted to and from it behind the scenes. Because the int data type is relied on so heavily for expression evaluations, the LP64 model is an ideal choice for 64-bit systems because it doesn’t change the int data type; as a result, it largely preserves the expected C type conversion behavior.

**Arithmetic Boundary Conditions**

You’ve learned that C’s basic integer types have minimum and maximum possible values determined by their underlying representation in memory. (Typical ranges for 32-bit twos complement architectures were presented in Table 6-2.) So, now you can explore what can happen when you attempt to traverse these boundaries. Simple arithmetic on a variable, such as addition, subtraction, or multiplication, can result in a value that can’t be held in that variable. Take a look at this example:

```c
unsigned int a;
a=0xe00000020;
a=a+0x200000020;
```

You know that a can hold a value of 0xE00000020 without a problem; Table 6-2 lists the maximum value of an unsigned 32-bit variable as 4,294,967,295, or 0xFFFF_FFFF. However, when 0x200000020 is added to 0xE0000000, the result, 0x100000040, can’t be held in a. When an arithmetic operation results in a value higher than the maximum possible representable value, it’s called a **numeric overflow condition**.
Here’s a slightly different example:

```c
unsigned int a;
a=0;
a=a-1;
```

The programmer subtracts 1 from `a`, which has an initial value of 0. The resulting value, -1, can’t be held in `a` because it’s below the minimum possible value of 0. This result is known as a **numeric underflow condition**.

**Note**

Numeric overflow conditions are also referred to in secure-programming literature as numeric overflows, arithmetic overflows, integer overflows, or integer wrapping. Numeric underflow conditions can be referred to as numeric underflows, arithmetic underflows, integer underflows, or integer wrapping. Specifically, the terms “wrapping around a value” or “wrapping below zero” might be used.

Although these conditions might seem as though they would be infrequent or inconsequential in real code, they actually occur quite often, and their impact can be quite severe from a security perspective. The incorrect result of an arithmetic operation can undermine the application’s integrity and often result in a compromise of its security. A numeric overflow or underflow that occurs early in a block of code can lead to a subtle series of cascading faults; not only is the result of a single arithmetic operation tainted, but every subsequent operation using that tainted result introduces a point where an attacker might have unexpected influence.

**Note**

Although numeric wrapping is common in most programming languages, it’s a particular problem in C/C++ programs because C requires programmers to perform low-level tasks that more abstracted high-level languages handle automatically. These tasks, such as dynamic memory allocation and buffer length tracking, often require arithmetic that might be vulnerable. Attackers commonly leverage arithmetic boundary conditions by manipulating a length calculation so that an insufficient amount of memory is allocated. If this happens, the program later runs the risk of manipulating memory outside the bounds of the allocated space, which often leads to an exploitable situation. Another common attack technique is bypassing a length check that protects sensitive operations, such as memory copies.
This chapter offers several examples of how underflow and overflow conditions lead to exploitable vulnerabilities. In general, auditors should be mindful of arithmetic boundary conditions when reviewing code and be sure to consider the possible implications of the subtle, cascading nature of these flaws.

In the following sections, you look at arithmetic boundary conditions affecting unsigned integers and then examine signed integers.

**Warning**

An effort has been made to use int and unsigned int types in examples to avoid code that’s affected by C’s default type promotions. This topic is covered in “Type Conversions” later in the chapter, but for now, note that whenever you use a char or short in an arithmetic expression in C, it’s converted to an int before the arithmetic is performed.

**Unsigned Integer Boundaries**

Unsigned integers are defined in the C specification as being subject to the rules of modular arithmetic (see the “Modular Arithmetic” sidebar). For an unsigned integer that uses X bits of storage, arithmetic on that integer is performed modulo 2^X. For example, arithmetic on a 8-bit unsigned integer is performed modulo 2^8, or modulo 256. Take another look at this simple expression:

```c
unsigned int a;
a=0xE0000020;
a=a+0x20000020;
```

The addition is performed modulo 2^{32}, or modulo 4,294,967,296 (0x100000000). The result of the addition is 0x40, which is (0xE0000020 + 0x20000020) modulo 0x100000000.

Another way to conceptualize it is to consider the extra bits of the result of a numeric overflow as being truncated. If you do the calculation 0xE0000020 + 0x20000020 in binary, you would have the following:

```
1110 0000 0000 0000 0000 0000 0010 0000
+ 0010 0000 0000 0000 0000 0000 0010 0000
= 1 0000 0000 0000 0000 0000 0000 0100 0000
```
The result you actually get in a is 0x40, which has a binary representation of 0000 0000 0000 0000 0000 0000 0100 0000.

Modular Arithmetic

Modular arithmetic is a system of arithmetic used heavily in computer science. The expression “X modulo Y” means “the remainder of X divided by Y.” For example, 100 modulo 11 is 1 because when 100 is divided by 11, the answer is 9 and the remainder is 1. The modulus operator in C is written as %. So in C, the expression (100 % 11) evaluates to 1, and the expression (100 / 11) evaluates to 9.

Modular arithmetic is useful for making sure a number is bounded within a certain range, and you often see it used for this purpose in hash tables. To explain, when you have X modulo Y, and X and Y are positive numbers, you know that the highest possible result is Y-1 and the lowest is 0. If you have a hash table of 100 buckets, and you need to map a hash to one of the buckets, you could do this:

```
struct bucket *buckets[100];
...
bucket = buckets[hash % 100];
```

To see how modular arithmetic works, look at a simple loop:

```
for (i=0; i<20; i++)
    printf("%d ", i % 6);
printf("\n");
```

The expression (i % 6) essentially bounds i to the range 0 to 5. As the program runs, it prints the following:

```
0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1
```

You can see that as i advanced from 0 to 19, i % 6 also advanced, but it wrapped back around to 0 every time it hit its maximum value of 5. As you move forward through the value, you wrap around the maximum value of 5. If you move backward through the values, you wrap “under” 0 to the maximum value of 5.

You can see that it’s the same as the result of the addition but without the highest bit. This isn’t far from what’s happening at the machine level. For example, Intel architectures have a carry flag (CF) that holds this highest bit. C doesn’t have a mechanism for allowing access to this flag, but depending on the underlying architecture, it could be checked via assembly code.

Here’s an example of a numeric overflow condition that occurs because of multiplication:
unsigned int a;
a=0xe0000020;
a=a*0x42;

Again, the arithmetic is performed modulo 0x100000000. The result of the multiplication is 0xC0000840, which is (0xE0000020 * 0x42) modulo 0x100000000. Here it is in binary:

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\times & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
\hline
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

The result you actually get in \(a\), 0xC0000840, has a binary representation of 1100 0000 0000 0000 0000 1000 0100 0000. Again, you can see how the higher bits that didn’t fit into the result were effectively truncated. At a machine level, often it’s possible to detect an overflow with integer multiplication as well as recover the high bits of a multiplication. For example, on Intel the imul instruction uses a destination object that’s twice the size of the source operands when multiplying, and it sets the flags OF (overflow) and CF (carry) if the result of the multiplication requires a width greater than the source operand. Some code even uses inline assembly to check for numeric overflow (discussed in the “Multiplication Overflows on Intel” sidebar later in this chapter).

You’ve seen examples of how arithmetic overflows could occur because of addition and multiplication. Another operator that can cause overflows is left shift, which, for this discussion, can be thought of as multiplication with 2. It behaves much the same as multiplication, so an example hasn’t been provided.

Now, you can look at some security exposures related to numeric overflow of unsigned integers. Listing 6-2 is a sanitized, edited version of an exploitable condition found recently in a client’s code.

Listing 6-2

Integer Overflow Example

```
u_char *make_table(unsigned int width, unsigned int height, u_char *init_row)
{
unsigned int n;
int i;
u_char *buf;

n = width * height;

buf = (char *)malloc(n);
if (!buf)
    return (NULL);
```
for (i=0; i< height; i++)
    memcpy(&buf[i*width], init_row, width);

return buf;
}

The purpose of the make_table() function is to take a width, a height, and an initial row and create a table in memory where each row is initialized to have the same contents as init_row. Assume that users have control over the dimensions of the new table: width and height. If they specify large dimensions, such as a width of 1,000,000, and a height of 3,000, the function attempts to call malloc() for 3,000,000,000 bytes. The allocation likely fails, and the calling function detects the error and handles it gracefully. However, users can cause an arithmetic overflow in the multiplication of width and height if they make the dimensions just a bit larger. This overflow is potentially exploitable because the allocation is done by multiplying width and height, but the actual array initialization is done with a for loop. So if users specify a width of 0x400 and a height of 0x1000001, the result of the multiplication is 0x400000400. This value, modulo 0x100000000, is 0x00000400, or 1024. So 1024 bytes would be allocated, but then the for loop would copy init_row roughly 16 million too many times. A clever attacker might be able to leverage this overflow to take control of the application, depending on the low-level details of the process’s runtime environment.

Take a look at a real-world vulnerability that’s similar to the previous example, found in the OpenSSH server. Listing 6-3 is from the OpenSSH 3.1 challenge-response authentication code: auth2-chall.c in the input_userauth_info_response() function.

**Listing 6-3**

*Challenge-Response Integer Overflow Example in OpenSSH 3.1*

```c
u_int nresp;
...
    nresp = packet_get_int();
    if (nresp > 0) {
        response = xmalloc(nresp * sizeof(char*));
        for (i = 0; i < nresp; i++)
            response[i] = packet_get_string(NULL);
    }
    packet_check_eom();
```

The nresp unsigned integer is user controlled, and its purpose is to tell the server how many responses to expect. It’s used to allocate the response[] array and fill it with network data. During the allocation of the response[] array in the call to xmalloc(), nresp is multiplied by sizeof(char *), which is typically 4 bytes. If users specify an nresp value that’s large enough, a numeric overflow could occur, and the
result of the multiplication could end up being a small number. For example, if
nresp has a value of 0x40000020, the result of the multiplication with 4 is 0x80.
Therefore, 0x80 bytes are allocated, but the following for loop attempts to retrieve
0x40000020 strings from the packet! This turned out to be a critical remotely
exploitable vulnerability.

Now turn your attention to numeric underflows. With unsigned integers,
subtractions can cause a value to wrap under the minimum representable value of
0. The result of an underflow is typically a large positive number because of the
modulus nature of unsigned integers. Here’s a brief example:

unsigned int a;
a=0x10;
a=a-0x30;

Look at the calculation in binary:

```
0000 0000 0000 0000 0000 0000 0001 0000
- 0000 0000 0000 0000 0000 0000 0011 0000
= 1111 1111 1111 1111 1111 1111 1110 0000
```

The result you get in a is the bit pattern for 0xfffffffffe0, which in twos comple-
ment representation is the correct negative value of -0x20. Recall that in modulus
arithmetic, if you advance past the maximum possible value, you wrap around to 0.
A similar phenomenon occurs if you go below the minimum possible value: You
wrap around to the highest possible value. Since a is an unsigned int type, it has a
value of 0xfffffffffe0 instead of -0x20 after the subtraction. Listing 6-4 is an example of
a numeric underflow involving an unsigned integer.

Listing 6-4

*Unsigned Integer Underflow Example*

```
struct header {
    unsigned int length;
    unsigned int message_type;
};

cchar *read_packet(int sockfd)
{
    int n;
    unsigned int length;
    struct header hdr;
    static char buffer[1024];

    if(full_read(sockfd, (void *)&hdr, sizeof(hdr))<=0){
        error("full_read: %m");
        return NULL;
    }
    return buffer;
```
This code reads a packet header from the network and extracts a 32-bit length field into the length variable. The length variable represents the total number of bytes in the packet, so the program first checks that the data portion of the packet isn’t longer than 1024 bytes to prevent an overflow. It then tries to read the rest of the packet from the network by reading \( (\text{length} - \text{sizeof(struct header)})\) bytes into buffer. This makes sense, as the code wants to read in the packet’s data portion, which is the total length minus the length of the header.

The vulnerability is that if users supply a length less than sizeof(struct header), the subtraction of \( (\text{length} - \text{sizeof(struct header)})\) causes an integer underflow and ends up passing a very large size parameter to full_read(). This error could result in a buffer overflow because at that point, read() would essentially copy data into the buffer until the connection is closed, which would allow attackers to take control of the process.

### Multiplication Overflows on Intel

Generally, processors detect when an integer overflow occurs and provide mechanisms for dealing with it; however, they are seldom used for error checking and generally aren’t accessible from C. For example, Intel processors set the overflow flag (OF) in the EFLAGS register when a multiplication causes an overflow, but a C programmer can’t check that flag without using inline assembler. Sometimes this is done for security reasons, such as the NDR unmarshalling routines for handling...
MSRPC requests in Windows operating systems. The following code, taken from \texttt{rpcrt4.dll}, is called when unmarshalling various data types from RPC requests:

```assembly
sub_77D6B6D4 proc near

var_of = dword ptr -4
arg_count = dword ptr 8
arg_length = dword ptr 0Ch
push ebp
mov ebp, esp
push ecx
and [ebp+var_of], 0
; set overflow flag to 0
push esi
mov esi, [ebp+arg_length]
imul esi, [ebp+arg_count]
; multiply length * count
jno short check_of
mov [ebp+var_of], 1
; if of set, set out flag
check_of:
cmp [ebp+var_of], 0
jnz short raise_ex
; must not overflow
cmp esi, 7FFFFFFFh
jbe short return
; must be a positive int
raise_ex:
push 6C6h
; exception
call RpcRaiseException
return:
mov eax, esi
; return result
pop esi
leave
retn 8
```

Arithmetic Boundary Conditions

continues...
Multiplication Overflows on Intel  Continued
You can see that this function, which multiplies the number of provided elements with the size of each element, does two sanity checks. First, it uses jno to check the overflow flag to make sure the multiplication didn’t overflow. Then it makes sure the resulting size is less than or equal to the maximum representable value of a signed integer, 0x7FFFFFFF. If either check fails, the function raises an exception.

Signed Integer Boundaries
Signed integers are a slightly different animal. According to the C specifications, the result of an arithmetic overflow or underflow with a signed integer is implementation defined and could potentially include a machine trap or fault. However, on most common architectures, the results of signed arithmetic overflows are well defined and predictable and don’t result in any kind of exception. These boundary behaviors are a natural consequence of how twos complement arithmetic is implemented at the hardware level, and they should be consistent on mainstream machines.

If you recall, the maximum positive value that can be represented in a twos complement signed integer is one in which all bits are set to 1 except the most significant bit, which is 0. This is because the highest bit indicates the sign of the number, and a value of 1 in that bit indicates that the number is negative. When an operation on a signed integer causes an arithmetic overflow or underflow to occur, the resulting value “wraps around the sign boundary” and typically causes a change in sign. For example, in a 32-bit integer, the value 0x7FFFFFFF is a large positive number. Adding 1 to it produces the result 0x80000000, which is a large negative number. Take a look at another simple example:

```c
int a;
a=0x7FFFFFFF0;
a=a+0x100;
```

The result of the addition is -0x7fffffff10, or -2,147,483,408. Now look at the calculation in binary:

```
0111 1111 1111 1111 1111 1111 1111 0000
+ 0000 0000 0000 0000 0000 0001 0000 0000
= 1000 0000 0000 0000 0000 0000 1111 0000
```

The result you get in a is the bit pattern for 0x800000f0, which is the correct result of the addition, but because it’s interpreted as a twos complement number, the value is actually interpreted as -0x7fffffff10. In this case, a large positive number plus a small positive number resulted in a large negative number.
With signed addition, you can overflow the sign boundary by causing a positive number to wrap around 0x80000000 and become a negative number. You can also underflow the sign boundary by causing a negative number to wrap below 0x80000000 and become a positive number. Subtraction is identical to addition with a negative number, so you can analyze them as being essentially the same operation. Overflows during multiplication and shifting are also possible, and classifying their results isn’t as easy. Essentially, the bits fall as they may; if a bit happens to end up in the sign bit of the result, the result is negative. Otherwise, it’s not. Arithmetic overflows involving multiplication seem a little tricky at first glance, but attackers can usually make them return useful, targeted values.

**Note**
Throughout this chapter, the `read()` function is used to demonstrate various forms of integer-related flaws. This is a bit of an oversimplification for the purposes of clarity, as many modern systems validate the length argument to `read()` at the system call level. These systems, which include BSDs and the newer Linux 2.6 kernel, check that this argument is less than or equal to the maximum value of a correspondingly sized signed integer, thus minimizing the risk of memory corruption.

Certain unexpected sign changes in arithmetic can lead to subtly exploitable conditions in code. These changes can cause programs to calculate space requirements incorrectly, leading to conditions similar to those that occur when crossing the maximum boundary for unsigned integers. Bugs of this nature typically occur in applications that perform arithmetic on integers taken directly from external sources, such as network data or files. Listing 6-5 is a simple example that shows how crossing the sign boundary can adversely affect an application.

**Listing 6-5**

*Signed Integer Vulnerability Example*

```c
char *read_data(int sockfd)
{
    char *buf;
    int length = network_get_int(sockfd);

    if (!(buf = (char *)malloc(MAXCHARS)))
        die("malloc: %m");

    if (length < 0 || length + 1 >= MAXCHARS){
        free(buf);
        die("bad length: %d", value);
    }
}
```

*Arithmetic Boundary Conditions*
if (read(sockfd, buf, length) <= 0) {
    free(buf);
    die("read: %m");
}

buf[value] = '\0';

return buf;

This example reads an integer from the network and performs some sanity checks on it. First, the length is checked to ensure that it’s greater than or equal to zero and, therefore, positive. Then the length is checked to ensure that it’s less than MAXCHARS. However, in the second part of the length check, 1 is added to the length. This opens an attack vector: A value of 0x7FFFFFFF passes the first check (because it’s greater than 0) and passes the second length check (as 0x7FFFFFFF + 1 is 0x80000000, which is a negative value). read() would then be called with an effectively unbounded length argument, leading to a potential buffer overflow situation.

This kind of mistake is easy to make when dealing with signed integers, and it can be equally challenging to spot. Protocols that allow users to specify integers directly are especially prone to this type of vulnerability. To examine this in practice, take a look at a real application that performs an unsafe calculation. The following vulnerability was in the OpenSSL 0.9.6 codebase related to processing Abstract Syntax Notation (ASN.1) encoded data. (ASN.1 is a language used for describing arbitrary messages to be sent between computers, which are encoded using BER, its basic encoding rules.) This encoding is a perfect candidate for a vulnerability of this nature because the protocol deals explicitly with 32-bit integers supplied by untrusted clients. Listing 6-6 is taken from crypto/asn1/a_d2i_fp.c—the ASN1_d2i_fp() function, which is responsible for reading ASN.1 objects from buffered IO (BIO) streams. This code has been edited for brevity.

Listing 6-6

Integer Sign Boundary Vulnerability Example in OpenSSL 0.9.6

c.inf=ASN1_get_object(&(c.p),&(c.slen),&(c.tag),&(c.xclass),
        len-off);

...
Type Conversions

This code is called in a loop for retrieving ASN.1 objects. The ASN1_get_object() function reads an object header that specifies the length of the next ASN.1 object. This length is placed in the signed integer c.slen, which is then assigned to want. The ASN.1 object function ensures that this number isn’t negative, so the highest value that can be placed in c.slen is 0x7FFFFFFF. At this point, len is the amount of data already read in to memory, and off is the offset in that data to the object being parsed. So, (len-off) is the amount of data read into memory that hasn’t yet been processed by the parser. If the code sees that the object is larger than the available unparsed data, it decides to allocate more space and read in the rest of the object.

The BUF_MEM_grow() function is called to allocate the required space in the memory buffer b; its second argument is a size parameter. The problem is that the len+want expression used for the second argument can be overflowed. Say that upon entering this code, len is 200 bytes, and off is 50. The attacker specifies an object size of 0x7FFFFFFF, which ends up in want. 0x7FFFFFFF is certainly larger than the 150 bytes of remaining data in memory, so the allocation code will be entered. want will be subtracted by 150 to reflect the amount of data already read in, giving it a value of 0x7FFFFF69. The call to BUF_MEM_grow() will ask for len+want bytes, or 0x7FFFFF69 + 200. This is 0x80000031, which is interpreted as a large negative number.

Internally, the BUF_MEM_grow() function does a comparison to check its length argument against how much space it has previously allocated. Because a negative number is less than the amount of memory it has already allocated, it assumes everything is fine. So the reallocation is bypassed, and arbitrary amounts of data can be copied into allocated heap data, with severe consequences.

Type Conversions

C is extremely flexible in handling the interaction of different data types. For example, with a few casts, you can easily multiply an unsigned character with a signed long integer, add it to a character pointer, and then pass the result to a function expecting a pointer to a structure. Programmers are used to this flexibility, so they tend to mix data types without worrying too much about what’s going on behind the scenes.
To deal with this flexibility, when the compiler needs to convert an object of one type into another type, it performs what’s known as a type conversion. There are two forms of type conversions: explicit type conversions, in which the programmer explicitly instructs the compiler to convert from one type to another by casting, and implicit type conversions, in which the compiler does “hidden” transformations of variables to make the program function as expected.

**Note**

You might see type conversions referred to as “type coercions” in programming-language literature; the terms are synonymous.

Often it’s surprising when you first learn how many implicit conversions occur behind the scenes in a typical C program. These automatic type conversions, known collectively as the default type conversions, occur almost magically when a programmer performs seemingly straightforward tasks, such as making a function call or comparing two numbers.

The vulnerabilities resulting from type conversions are often fascinating, because they can be subtle and difficult to locate in source code, and they often lead to situations in which the patch for a critical remote vulnerability is as simple as changing a char to an unsigned char. The rules governing these conversions are deceptively subtle, and it’s easy to believe you have a solid grasp of them and yet miss an important nuance that makes a world of difference when you analyze or write code.

Instead of jumping right into known vulnerability classes, first you look at how C compilers perform type conversions at a low level, and then you study the rules of C in detail to learn about all the situations in which conversions take place. This section is fairly long because you have to cover a lot of ground before you have the foundation to analyze C’s type conversions with confidence. However, this aspect of the language is subtle enough that it’s definitely worth taking the time to gain a solid understanding of the ground rules; you can leverage this understanding to find vulnerabilities that most programmers aren’t aware of, even at a conceptual level.

**Overview**

When faced with the general problem of reconciling two different types, C goes to great lengths to avoid surprising programmers. The compilers follow a set of rules that attempt to encapsulate “common sense” about how to manage mixing different types, and more often than not, the result is a program that makes sense and simply does what the programmer intended. That said, applying these rules can often lead to surprising, unexpected behaviors. Moreover, as you might expect, these unexpected behaviors tend to have dire security consequences.
You start in the next section by exploring the conversion rules, the general rules C uses when converting between types. They dictate how a machine converts from one type to another type at the bit level. After you have a good grasp of how C converts between different types at the machine level, you examine how the compiler chooses which type conversions to apply in the context of C expressions, which involves three important concepts: simple conversions, integer promotions, and usual arithmetic conversions.

**Note**
Although non-integer types, such as floats and pointers, have some coverage, the primary focus of this discussion is on how C manipulates integers because these conversions are widely misunderstood and are critical for security analysis.

**Conversion Rules**
The following rules describe how C converts from one type to another, but they don’t describe when conversions are performed or why they are performed.

**Note**
The following content is specific to two's complement implementations and represents a distilled and pragmatic version of the rules in the C specification.

**Integer Types: Value Preservation**
An important concept in integer type conversions is the notion of a value-preserving conversion. Basically, if the new type can represent all possible values of the old type, the conversion is said to be value-preserving. In this situation, there’s no way the value can be lost or changed as a result of the conversion. For example, if an unsigned char is converted into an int, the conversion is value-preserving because an int can represent all of the values of an unsigned char. You can verify this by referring to Table 6-2 again. Assuming you’re considering a two's complement machine, you know that an 8-bit unsigned char can represent any value between 0 and 255. You know that a 32-bit int can represent any value between -2147483648 and 2147483647. Therefore, there’s no value the unsigned char can have that the int can’t represent.

Correspondingly, in a value-changing conversion, the old type can contain values that can’t be represented in the new type. For example, if you convert an int into an unsigned int, you have potentially created an intractable situation. The unsigned
int, on a 32-bit machine, has a range of 0 to 4294967295, and the int has a range of -2147483648 to 2147483647. The unsigned int can’t hold any of the negative values a signed int can represent.

According to the C standard, some of the value-changing conversions have implementation-defined results. This is true only for value-changing conversions that have a signed destination type; value-changing conversions to an unsigned type are defined and consistent across all implementations. (If you recall from the boundary condition discussion, this is because unsigned arithmetic is defined as a modulus arithmetic system.) Twos complement machines follow the same basic behaviors, so you can explain how they perform value-changing conversions to signed destination types with a fair amount of confidence.

Integer Types: Widening

When you convert from a narrow type to a wider type, the machine typically copies the bit pattern from the old variable to the new variable, and then sets all the remaining high bits in the new variable to 0 or 1. If the source type is unsigned, the machine uses zero extension, in which it propagates the value 0 to all high bits in the new wider type. If the source type is signed, the machine uses sign extension, in which it propagates the sign bit from the source type to all unused bits in the destination type.

Warning

The widening procedure might have some unexpected implications:
If a narrow signed type, such as signed char, is converted to a wider unsigned type, such as unsigned int, sign extension occurs.

Figure 6-1 shows a value-preserving conversion of an unsigned char with a value of 5 to a signed int.

![Figure 6-1](image)

The character is placed into the integer, and the value is preserved. At the bit pattern level, this simply involved zero extension: clearing out the high bits and moving the least significant byte (LSB) into the new object’s LSB.
Now consider a signed char being converted into a int. A int can represent all the values of a signed char, so this conversion is also value-preserving. Figure 6-2 shows what this conversion looks like at the bit level.

This situation is slightly different, as the value is the same, but the transformation is more involved. The bit representation of -5 in a signed char is 1111 1011. The bit representation of -5 in an int is 1111 1111 1111 1111 1111 1111 1111 1011. To do the conversion, the compiler generates assembly that performs sign extension. You can see in Figure 6-2 that the sign bit is set in the signed char, so to preserve the value -5, the sign bit has to be copied to the other 24 bits of the int.

The previous examples are value-preserving conversions. Now consider a value-changing widening conversion. Say you convert a signed char with a value of -5 to an unsigned int. Because the source type is signed, you perform sign extension on the signed char before placing it in the unsigned int (see Figure 6-3).

As mentioned previously, this result can be surprising to developers. You explore its security ramifications in “Sign Extension” later in this chapter. This conversion is value changing because an unsigned int can’t represent values less than 0.

**Integer Types: Narrowing**

When converting from a wider type to a narrower type, the machine uses only one mechanism: **truncation**. The bits from the wider type that don’t fit in the new narrower type are dropped. Figures 6-4 and 6-5 show two narrowing conversions. Note that all narrowing conversions are value-changing because you’re losing precision.
One final type of integer conversion to consider: If a conversion occurs between a signed type and an unsigned type of the same width, nothing is changed in the bit pattern. This conversion is value-changing. For example, say you have the signed integer -1, which is represented in binary as 1111 1111 1111 1111 1111 1111 1111 1111.

If you interpret this same bit pattern as an unsigned integer, you see a value of 4,294,967,295. The conversion is summarized in Figure 6-6. The conversion from unsigned int to int technically might be implementation defined, but it works in the same fashion: The bit pattern is left alone, and the value is interpreted in the context of the new type (see Figure 6-7).
Integer Type Conversion Summary

Here are some practical rules of thumb for integer type conversions:

- When you convert from a narrower signed type to a wider unsigned type, the compiler emits assembly to do sign extension, and the value of the object might change.
- When you convert from a narrower signed type to a wider signed type, the compiler emits assembly to do sign extension, and the value of the object is preserved.
- When you convert from a narrower unsigned type to a wider type, the compiler emits assembly to do zero extension, and the value of the object is preserved.
- When you convert from a wider type to a narrower type, the compiler emits assembly to do truncation, and the value of the object might change.
- When you convert between signed and unsigned types of the same width, the compiler effectively does nothing, the bit pattern stays the same, and the value of the object might change.

Table 6-4 summarizes the processing that occurs when different integer types are converted in two's complement implementations of C. As you cover the information in the following sections, this table can serve as a useful reference for recalling how a conversion occurs.

**Table 6-4**

| Integer Type Conversion in C (Source on Left, Destination on Top) |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| signed char | unsigned char | short int | Unsigned short int | int | unsigned int |
| signed char | Compatible types | Value changing Bit pattern same | Value preserving Sign extension | Value changing Sign extension | Value preserving Sign extension | Value changing Sign extension |

Figure 6-7  Conversion of unsigned int to signed int (big endian)
### Table 6-4  continued

<table>
<thead>
<tr>
<th>Integer Type Conversion in C (Source on Left, Destination on Top)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>signed char</strong></td>
</tr>
<tr>
<td>Value changing</td>
</tr>
<tr>
<td>Bit pattern same</td>
</tr>
<tr>
<td>Implementation defined</td>
</tr>
</tbody>
</table>

#### Floating Point and Complex Types

Although vulnerabilities caused by the use of floating point arithmetic haven’t been widely published, they are certainly possible. There’s certainly the possibility of subtle errors surfacing in financial software related to floating point type conversions or representation issues. The discussion of floating point types in this chapter is fairly brief. For more information, refer to the C standards documents and the previously mentioned C programming references.

The C standard’s rules for conversions between real floating types and integer types leave a lot of room for implementation-defined behaviors. In a conversion from a real type to an integer type, the fractional portion of the number is
discarded. If the integer type can't represent the integer portion of the floating point number, the result is undefined. Similarly, a conversion from an integer type to a real type transfers the value over if possible. If the real type can't represent the integer's value but can come close, the compiler rounds the integer to the next highest or lowest number in an implementation-defined manner. If the integer is outside the range of the real type, the result is undefined.

Conversions between floating point types of different precision are handled with similar logic. Promotion causes no change in value. During a demotion that causes a change in value, the compiler is free to round numbers, if possible, in an implementation-defined manner. If rounding isn't possible because of the range of the target type, the result is undefined.

Other Types
There are myriad other types in C beyond integers and floats, including pointers, Booleans, structures, unions, functions, arrays, enums, and more. For the most part, conversion among these types isn't quite as critical from a security perspective, so they aren't extensively covered in this chapter.

Pointer arithmetic is covered in “Pointer Arithmetic” later in this chapter. Pointer type conversion depends largely on the underlying machine architecture, and many conversions are specified as implementation defined. Essentially, programmers are free to convert pointers into integers and back, and convert pointers from one type to another. The results are implementation defined, and programmers need to be cognizant of alignment restrictions and other low-level details.

Simple Conversions
Now that you have a good idea how C converts from one integer type to another, you can look at some situations where these type conversions occur. Simple conversions are C expressions that use straightforward applications of conversion rules.

Casts
As you know, typecasts are C’s mechanism for letting programmers specify an explicit type conversion, as shown in this example:

(unsigned char) bob

Whatever type bob happens to be, this expression converts it into an unsigned char type. The resulting type of the expression is unsigned char.

Assignments
Simple type conversion also occurs in the assignment operator. The compiler must convert the type of the right operand into the type of the left operand, as shown in this example:
short int fred;
int bob = -10;

fred = bob;

For both assignments, the compiler must take the object in the right operand and convert it into the type of the left operand. The conversion rules tell you that conversion from the int bob to the short int fred results in truncation.

Function Calls: Prototypes
C has two styles of function declarations: the old K&R style, in which parameter types aren’t specified in the function declaration, and the new ANSI style, in which the parameter types are part of the declaration. In the ANSI style, the use of function prototypes is still optional, but it’s common. With the ANSI style, you typically see something like this:

int dostuff(int jim, unsigned char bob);

void func(void)
{
    char a=42;
    unsigned short b=43;
    long long int c;

    c=dostuff(a, b);
}

The function declaration for dostuff() contains a prototype that tells the compiler the number of arguments and their types. The rule of thumb is that if the function has a prototype, the types are converted in a straightforward fashion using the rules documented previously. If the function doesn’t have a prototype, something called the default argument promotions kicks in (explained in “Integer Promotions”). The previous example has a character (a) being converted into an int (jim), an unsigned short (b) being converted into an unsigned char (bob), and an int (the dostuff() function’s return value) being converted into a long long int (c).

Function Calls: return
return does a conversion of its operand to the type specified in the enclosing function’s definition. For example, the int a is converted into a char data type by return:
char func(void)
{
    int a=42;
    return a;
}

Integer Promotions

Integer promotions specify how C takes a narrow integer data type, such as a char or short, and converts it to an int (or, in rare cases, to an unsigned int). This up-conversion, or promotion, is used for two different purposes:

- Certain operators in C require an integer operand of type int or unsigned int. For these operators, C uses the integer promotion rules to transform a narrower integer operand into the correct type—int or unsigned int.
- Integer promotions are a critical component of C’s rules for handling arithmetic expressions, which are called the usual arithmetic conversions. For arithmetic expressions involving integers, integer promotions are usually applied to both operands.

**Note**
You might see the terms “integer promotions” and “integral promotions” used interchangeably in other literature, as they are synonymous.

There’s a useful concept from the C standards: Each integer data type is assigned what’s known as an integer conversion rank. These ranks order the integer data types by their width from lowest to highest. The signed and unsigned varieties of each type are assigned the same rank. The following abridged list sorts integer types by conversion rank from high to low. The C standard assigns ranks to other integer types, but this list should suffice for this discussion:

- long long int, unsigned long long int
- long int, unsigned long int
- unsigned int, int
- unsigned short, short
- char, unsigned char, signed char
- _Bool
Basically, any place in C where you can use an int or unsigned int, you can also use any integer type with a lower integer conversion rank. This means you can use smaller types, such as chars and short ints, in the place of ints in C expressions. You can also use a bit field of type _Bool, int, signed int, or unsigned int. The bit fields aren’t ascribed integer conversion ranks, but they are treated as narrower than their corresponding base type. This makes sense because a bit field of an int is usually smaller than an int, and at its widest, it’s the same width as an int.

If you apply the integer promotions to a variable, what happens? First, if the variable isn’t an integer type or a bit field, the promotions do nothing. Second, if the variable is an integer type, but its integer conversion rank is greater than or equal to that of an int, the promotions do nothing. Therefore, ints, unsigned ints, long ints, pointers, and floats don’t get altered by the integer promotions.

So, the integer promotions are responsible for taking a narrower integer type or bit field and promoting it to an int or unsigned int. This is done in a straightforward fashion: If a value-preserving transformation to an int can be performed, it’s done. Otherwise, a value-preserving conversion to an unsigned int is performed.

In practice, this means almost everything is converted to an int, as an int can hold the minimum and maximum values of all the smaller types. The only types that might be promoted to an unsigned int are unsigned int bit fields with 32 bits or perhaps some implementation-specific extended integer types.

**Historical Note**

The C89 standard made an important change to the C type conversion rules. In the K&R days of the C language, integer promotions were **unsigned-preserving** rather than value-preserving. So with the current C rules, if a narrower, unsigned integer type, such as an unsigned char, is promoted to a wider, signed integer, such as an int, value conversion dictates that the new type is a signed integer. With the old rules, the promotion would preserve the unsigned-ness, so the resulting type would be an unsigned int. This changed the behavior of many signed/unsigned comparisons that involved promotions of types narrower than int.

**Integer Promotions Summary**

The basic rule of thumb is this: If an integer type is narrower than an int, integer promotions almost always convert it to an int. Table 6-5 summarizes the result of integer promotions on a few common types.
Table 6-5

Results of Integer Promotions

<table>
<thead>
<tr>
<th>Source Type</th>
<th>Result Type</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char</td>
<td>int</td>
<td>Promote; source rank less than int rank</td>
</tr>
<tr>
<td>char</td>
<td>int</td>
<td>Promote; source rank less than int rank</td>
</tr>
<tr>
<td>short</td>
<td>int</td>
<td>Promote; source rank less than int rank</td>
</tr>
<tr>
<td>unsigned short</td>
<td>int</td>
<td>Promote; source rank less than int rank</td>
</tr>
<tr>
<td>unsigned int: 24</td>
<td>int</td>
<td>Promote; bit field of unsigned int</td>
</tr>
<tr>
<td>unsigned int: 32</td>
<td>unsigned int</td>
<td>Promote; bit field of unsigned int</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>Don’t promote; source rank equal to int rank</td>
</tr>
<tr>
<td>unsigned int</td>
<td>unsigned int</td>
<td>Don’t promote; source rank equal to int rank</td>
</tr>
<tr>
<td>long int</td>
<td>long int</td>
<td>Don’t promote; source rank greater than int rank</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>Don’t promote; source not of integer type</td>
</tr>
<tr>
<td>char *</td>
<td>char *</td>
<td>Don’t promote; source not of integer type</td>
</tr>
</tbody>
</table>

Integer Promotion Applications

Now that you understand integer promotions, the following sections examine where they are used in the C language.

Unary + Operator

The unary + operator performs integer promotions on its operand. For example, if the bob variable is of type char, the resulting type of the expression (+bob) is int, whereas the resulting type of the expression (bob) is char.

Unary - Operator

The unary - operator does integer promotion on its operand and then does a negation. Regardless of whether the operand is signed after the promotion, a twos complement negation is performed, which involves inverting the bits and adding 1.

The Leblancian Paradox

David Leblanc is an accomplished researcher and author, and one of the world’s foremost experts on integer issues in C and C++. He documented a fascinating nuance of twos complement arithmetic that he discovered while working on the SafeInt class with his colleague Atin Bansal (http://msdn.microsoft.com/library/en-us/dncode/html/secure01142004.asp). To negate a twos complement number, you flip all the bits and add 1 to the result. Assuming a 32-bit signed data type, what’s the inverse of 0x80000000?
The Leblancian Paradox  Continued

If you flip all the bits, you get 0x7fffffff. If you add 1, you get 0x80000000. So the unary negation of this corner-case number is itself!

This idiosyncrasy can come into play when developers use negative integers to represent a special sentinel set of numbers or attempt to take the absolute value of an integer. In the following code, the intent is for a negative index to specify a secondary hash table. This works fine unless attackers can specify an index of 0x80000000. The negation of the number results in no change in the value, and 0x80000000 % 1000 is -648, which causes memory before the array to be modified.

```c
int bank1[1000], bank2[1000];
...
void hashbank(int index, int value)
{
    int *bank = bank1;

    if (index<0) {
        bank = bank2;
        index = -index;
    }

    bank[index % 1000] = value;
}
```

Unary ~ Operator

The unary ~ operator does a ones complement of its operand after doing an integer promotion of its operand. This effectively performs the same operation on both signed and unsigned operands for two's complement implementations: It inverts the bits.

Bitwise Shift Operators

The bitwise shift operators >> and << shift the bit patterns of variables. The integer promotions are applied to both arguments of these operators, and the type of the result is the same as the promoted type of the left operand, as shown in this example:
char a = 1;
char c = 16;
int bob;
bob = a << c;

a is converted to an integer, and c is converted to an integer. The promoted type of the left operand is int, so the type of the result is an int. The integer representation of a is left-shifted 16 times.

**Switch Statements**

Integer promotions are used in switch statements. The general form of a switch statement is something like this:

```c
switch (controlling expression)
{
    case (constant integer expression): body;
        break;
    default: body;
        break;
}
```

The integer promotions are used in the following way: First, they are applied to the controlling expression, so that expression has a promoted type. Then, all the integer constants are converted to the type of the promoted control expression.

**Function Invocations**

Older C programs using the K&R semantics don’t specify the data types of arguments in their function declarations. When a function is called without a prototype, the compiler has to do something called **default argument promotions**. Basically, integer promotions are applied to each function argument, and any arguments of the float type are converted to arguments of the double type. Consider the following example:

```c
int jim(char bob)
{
    printf("bob=%d\n", bob);
}

int main(int argc, char **argv)
{
```
In this example, a copy of the value of a is passed to the jim() function. The char type is first run through the integer promotions and transformed into an integer. This integer is what’s passed to the jim() function. The code the compiler emits for the jim() function is expecting an integer argument, and it performs a direct conversion of that integer back into a char format for the bob variable.

Usual Arithmetic Conversions

In many situations, C is expected to take two operands of potentially divergent types and perform some arithmetic operation that involves both of them. The C standards spell out a general algorithm for reconciling two types into a compatible type for this purpose. This procedure is known as the usual arithmetic conversions. The goal of these conversions is to transform both operands into a common real type, which is used for the actual operation and then as the type of the result. These conversions apply only to the arithmetic types—integer and floating point types. The following sections tackle the conversion rules.

Rule 1: Floating Points Take Precedence

Floating point types take precedence over integer types, so if one of the arguments in an arithmetic expression is a floating point type, the other argument is converted to a floating point type. If one floating point argument is less precise than the other, the less precise argument is promoted to the type of the more precise argument.

Rule 2: Apply Integer Promotions

If you have two operands and neither is a float, you get into the rules for reconciling integers. First, integer promotions are performed on both operands. This is an extremely important piece of the puzzle! If you recall from the previous section, this means any integer type smaller than an int is converted into an int, and anything that’s the same width as an int, larger than an int, or not an integer type is left alone. Here’s a brief example:

```c
unsigned char jim = 255;
unsigned char bob = 255;
if ((jim + bob) > 300) do_something();
```

In this expression, the + operator causes the usual arithmetic conversions to be applied to its operands. This means both jim and bob are promoted to ints, the
addition takes place, and the resulting type of the expression is an int that holds the result of the addition (510). Therefore, do_something() is called, even though it looks like the addition could cause a numeric overflow. To summarize: Whenever there’s arithmetic involving types narrower than an integer, the narrow types are promoted to integers behind the scenes. Here’s another brief example:

```c
unsigned short a=1;
if ((a-5) < 0) do_something();
```

Intuition would suggest that if you have an unsigned short with the value 1, and it’s subtracted by 5, it underflows around 0 and ends up containing a large value. However, if you test this fragment, you see that do_something() is called because both operands of the subtraction operator are converted to ints before the comparison. So a is converted from an unsigned short to an int, and then an int with a value of 5 is subtracted from it. The resulting value is -4, which is a valid integer value, so the comparison is true. Note that if you did the following, do_something() wouldn’t be called:

```c
unsigned short a=1;
a=a-5;
if (a < 0) do_something();
```

The integer promotion still occurs with the (a-5), but the resulting integer value of -4 is placed back into the unsigned short a. As you know, this causes a simple conversion from signed int to unsigned short, which causes truncation to occur, and a ends up with a large positive value. Therefore, the comparison doesn’t succeed.

**Rule 3: Same Type After Integer Promotions**

If the two operands are of the same type after integer promotions are applied, you don’t need any further conversions because the arithmetic should be straightforward to carry out at the machine level. This can happen if both operands have been promoted to an int by integer promotions, or if they just happen to be of the same type and weren’t affected by integer promotions.

**Rule 4: Same Sign, Different Types**

If the two operands have different types after integer promotions are applied, but they share the same signed-ness, the narrower type is converted to the type of the wider type. In other words, if both operands are signed or both operands are unsigned, the type with the lesser integer conversion rank is converted to the type of the operand with the higher conversion rank.

Note that this rule has nothing to do with short integers or characters because they have already been converted to integers by integer promotions. This rule is
more applicable to arithmetic involving types of larger sizes, such as long long int or long int. Here’s a brief example:

```c
int jim = 5;
long int bob = 6;
long long int fred;
fred = (jim + bob)
```

Integer promotions don’t change any types because they are of equal or higher width than the int type. So this rule mandates that the int `jim` be converted into a long int before the addition occurs. The resulting type, a long int, is converted into a long long int by the assignment to `fred`.

In the next section, you consider operands of different types, in which one is signed and the other is unsigned, which gets interesting from a security perspective.

**Rule 5: Unsigned Type Wider Than or Same Width as Signed Type**

The first rule for this situation is that if the unsigned operand is of greater integer conversion rank than the signed operand, or their ranks are equal, you convert the signed operand to the type of the unsigned operand. This behavior can be surprising, as it leads to situations like this:

```c
int jim = -5;
if (jim < sizeof (int))
    do_something();
```

The comparison operator `<` causes the usual arithmetic conversions to be applied to both operands. Integer promotions are applied to `jim` and to `sizeof(int)`, but they don’t affect them. Then you continue into the usual arithmetic conversions and attempt to figure out which type should be the common type for the comparison. In this case, `jim` is a signed integer, and `sizeof (int)` is a `size_t`, which is an unsigned integer type. Because `size_t` has a greater integer conversion rank, the unsigned type takes precedence by this rule. Therefore, `jim` is converted to an unsigned integer type, the comparison fails, and `do_something()` isn’t called. On a 32-bit system, the actual comparison is as follows:

```c
if (4294967291 < 4)
    do_something();
```

**Rule 6: Signed Type Wider Than Unsigned Type, Value Preservation Possible**

If the signed operand is of greater integer conversion rank than the unsigned operand, and a value-preserving conversion can be made from the unsigned integer type to the signed integer type, you choose to transform everything to the signed integer type, as in this example:
long long int a=10;
unsigned int b= 5;
(a+b);

The signed argument, a long long int, can represent all the values of the unsigned argument, an unsigned int, so the compiler would convert both operands to the signed operand’s type: long long int.

Rule 7: Signed Type Wider Than Unsigned Type, Value Preservation Impossible
There’s one more rule: If the signed integer type has a greater integer conversion rank than the unsigned integer type, but all values of the unsigned integer type can’t be held in the signed integer type, you have to do something a little strange. You take the type of the signed integer type, convert it to its corresponding unsigned integer type, and then convert both operands to use that type. Here’s an example:

unsigned int a = 10;
long int b=20;
(a+b);

For the purpose of this example, assume that on this machine, the long int size has the same width as the int size. The addition operator causes the usual arithmetic conversions to be applied. Integer promotions are applied, but they don’t change the types. The signed type (long int) is of higher rank than the unsigned type (unsigned int). The signed type (long int) can’t hold all the values of the unsigned type (unsigned int), so you’re left with the last rule. You take the type of the signed operand, which is a long int, convert it into its corresponding unsigned equivalent, unsigned long int, and then convert both operands to unsigned long int. The addition expression, therefore, has a resulting type of unsigned long int and a value of 30.

Summary of Arithmetic Conversions
The following is a summary of the usual arithmetic conversions. Table 6-6 demonstrates some sample applications of the usual arithmetic conversions.

- If either operand is a floating point number, convert all operands to the floating point type of the highest precision operand. You’re finished.
- Perform integer promotions on both operands. If the two operands are now of the same type, you’re finished.
- If the two operands share the same signed-ness, convert the operand with the lower integer conversion rank to the type of the operand of the higher integer conversion rank. You’re finished.
If the unsigned operand is of higher or equal integer conversion rank than the signed operand, convert the signed operand to the type of the unsigned operand. You’re finished.

If the signed operand is of higher integer conversion rank than the unsigned operand, and you can perform a value-preserving conversion, convert the unsigned operand to the signed operand’s type. You’re finished.

If the signed operand is of higher integer conversion rank than the unsigned operand, but you can’t perform a value-preserving conversion, convert both operands to the unsigned type that corresponds to the type of the signed operand.

Table 6-6

<table>
<thead>
<tr>
<th>Left Operand Type</th>
<th>Right Operand Type</th>
<th>Result</th>
<th>Common Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>float</td>
<td>1. Left operand converted to float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>char</td>
<td>1. Right operand converted to double</td>
<td>double</td>
</tr>
<tr>
<td>unsigned int</td>
<td>int</td>
<td>1. Right operand converted to unsigned int</td>
<td>unsigned int</td>
</tr>
<tr>
<td>unsigned short</td>
<td>int</td>
<td>1. Left operand converted to int</td>
<td>int</td>
</tr>
<tr>
<td>unsigned char</td>
<td>unsigned short</td>
<td>1. Left operand converted to int 2. Right operand converted to int</td>
<td>int</td>
</tr>
<tr>
<td>unsigned int: 32</td>
<td>short</td>
<td>1. Left operand converted to unsigned int 2. Right operand converted to int 3. Right operand converted to unsigned int</td>
<td>unsigned int</td>
</tr>
<tr>
<td>unsigned int</td>
<td>long int</td>
<td>1. Left operand converted to unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>unsigned int</td>
<td>long long int</td>
<td>1. Left operand converted to long long int</td>
<td>long long int</td>
</tr>
<tr>
<td>unsigned int</td>
<td>unsigned long long int</td>
<td>1. Left operand converted to unsigned long long int</td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>

Usual Arithmetic Conversion Applications

Now that you have a grasp of the usual arithmetic conversions, you can look at where these conversions are used.
Addition
Addition can occur between two arithmetic types as well as between a pointer type and an arithmetic type. Pointer arithmetic is explained in “Pointer Arithmetic,” but for now, you just need to note that when both arguments are an arithmetic type, the compiler applies the usual arithmetic conversions to them.

Subtraction
There are three types of subtraction: subtraction between two arithmetic types, subtraction between a pointer and an arithmetic type, and subtraction between two pointer types. In subtraction between two arithmetic types, C applies the usual arithmetic conversions to both operands.

Multiplicative Operators
The operands to * and / must be an arithmetic type, and the arguments to % must be an integer type. The usual arithmetic conversions are applied to both operands of these operators.

Relational and Equality Operators
When two arithmetic operands are compared, the usual arithmetic conversions are applied to both operands. The resulting type is an int, and its value is 1 or 0, depending on the result of the test.

Binary Bitwise Operators
The binary bitwise operators &, ^, and | require integer operands. The usual arithmetic conversions are applied to both operands.

Question Mark Operator
From a type conversion perspective, the conditional operator is one of C’s more interesting operators. Here’s a short example of how it’s commonly used:

```c
int a=1;
unsigned int b=2;
int choice=-1;
...
result = choice ? a : b ;
```

In this example, the first operand, choice, is evaluated as a scalar. If it’s set, the result of the expression is the evaluation of the second operand, which is a. If it’s not set, the result is the evaluation of the third operand, b.

The compiler has to know at compile time the result type of the conditional expression, which could be tricky in this situation. What C does is determine which type would be the result of running the usual arithmetic conversions against the
second and third arguments, and it makes that type the resulting type of the expression. So in the previous example, the expression results in an unsigned int, regardless of the value of choice.

**Type Conversion Summary**

Table 6-7 shows the details of some common type conversions.

**Table 6-7**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand Types</th>
<th>Conversions</th>
<th>Resulting Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typecast</td>
<td></td>
<td>Expression is converted to type using simple conversions</td>
<td>Type</td>
</tr>
<tr>
<td>(type)expression</td>
<td></td>
<td>Right operand converted to left operand type using simple conversions</td>
<td>Type of left operand</td>
</tr>
<tr>
<td>Assignment =</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function call with prototype</td>
<td>Arguments converted using simple conversions according to prototype</td>
<td>Return type of function</td>
<td></td>
</tr>
<tr>
<td>Function call without prototype</td>
<td>Arguments promoted via default argument promotions, which are essentially integer promotions</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>Return</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unary +, -</td>
<td>Operand must be arithmetic type</td>
<td>Operand undergoes integer promotions</td>
<td>Promoted type of operand</td>
</tr>
<tr>
<td>+a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unary ~</td>
<td>Operand must be integer type</td>
<td>Operand undergoes integer promotions</td>
<td>Promoted type of operand</td>
</tr>
<tr>
<td>~a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitwise &lt;&lt; and &gt;&gt;</td>
<td>Operands must be integer type</td>
<td>Operands undergo integer promotions</td>
<td>Promoted type of left operand</td>
</tr>
<tr>
<td>switch statement</td>
<td>Expression must have integer type</td>
<td>Expression undergoes integer promotion; cases are converted to that type</td>
<td></td>
</tr>
<tr>
<td>Binary +, -</td>
<td>Operands must be arithmetic type</td>
<td>Common type from usual arithmetic conversions</td>
<td></td>
</tr>
</tbody>
</table>

*Pointer arithmetic covered in "Pointer Arithmetic"*
## Type Conversions

<table>
<thead>
<tr>
<th>Binary <code>*</code> and <code>/</code></th>
<th>Operands must be arithmetic type</th>
<th>Operands undergo usual arithmetic conversions</th>
<th>Common type from usual arithmetic conversions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary <code>%</code></td>
<td>Operands must be integer type</td>
<td>Operands undergo usual arithmetic conversions</td>
<td>Common type from usual arithmetic conversions</td>
</tr>
<tr>
<td>Binary subscript <code>[]</code></td>
<td><code>a[b]</code></td>
<td>Interpreted as <code>*((a)+(b))</code></td>
<td>int, value 0 or 1</td>
</tr>
<tr>
<td>Unary <code>!</code></td>
<td>Operand must be arithmetic type or pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sizeof</code></td>
<td></td>
<td><code>size_t</code> (unsigned integer type)</td>
<td></td>
</tr>
<tr>
<td>Binary <code>&lt; &gt; &lt;= =&gt; == !=</code></td>
<td>Operands must be arithmetic type</td>
<td>Operands undergo usual arithmetic conversions</td>
<td>int, value 0 or 1</td>
</tr>
<tr>
<td>`&amp; ^</td>
<td>`</td>
<td>Operands must be integer type</td>
<td>Operands undergo usual arithmetic conversions</td>
</tr>
<tr>
<td>`&amp;&amp;</td>
<td></td>
<td>`</td>
<td>Operands must be arithmetic type or pointer</td>
</tr>
<tr>
<td>Conditional <code>?</code></td>
<td>2nd and 3rd operands must be arithmetic type or pointer</td>
<td>Second and third operands undergo usual arithmetic conversions</td>
<td>Common type from usual arithmetic conversions</td>
</tr>
<tr>
<td></td>
<td>Type of right operand</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Auditing Tip: Type Conversions

Even those who have studied conversions extensively might still be surprised at the way a compiler renders certain expressions into assembly. When you see code that strikes you as suspicious or potentially ambiguous, never hesitate to write a simple test program or study the generated assembly to verify your intuition.

If you do generate assembly to verify or explore the conversions discussed in this chapter, be aware that C compilers can optimize out certain conversions or use architectural tricks that might make the assembly appear incorrect or inconsistent. At a conceptual level,
compilers are behaving as the C standard describes, and they ultimately generate code that follows the rules. However, the assembly might look inconsistent because of optimizations or even incorrect, as it might manipulate portions of registers that should be unused.

Type Conversion Vulnerabilities

Now that you have a solid grasp of C’s type conversions, you can explore some of the exceptional circumstances they can create. Implicit type conversions can catch programmers off-guard in several situations. This section focuses on simple conversions between signed and unsigned types, sign extension, truncation, and the usual arithmetic conversions, focusing on comparisons.

Signed/Unsigned Conversions

Most security issues related to type conversions are the result of simple conversions between signed and unsigned integers. This discussion is limited to conversions that occur as a result of assignment, function calls, or typecasts.

For a quick recap of the simple conversion rules, when a signed variable is converted to an unsigned variable of the same size, the bit pattern is left alone, and the value changes correspondingly. The same thing occurs when an unsigned variable is converted to a signed variable. Technically, the unsigned-to-signed conversion is implementation defined, but in two's complement implementations, usually the bit pattern is left alone.

The most important situation in which this conversion becomes relevant is during function calls, as shown in this example:

```c
int copy(char *dst, char *src, unsigned int len)
{
    while (len--)
        *dst++ = *src++;
}
```

The third argument is an unsigned int that represents the length of the memory section to copy. If you call this function and pass a signed int as the third argument, it’s converted to an unsigned integer. For example, say you do this:

```c
int f = -1;
copy(mydst, mysrc, f);
```
The copy() function sees an extremely large positive len and most likely copies until it causes a segmentation fault. Most libc routines that take a size parameter have an argument of type size_t, which is an unsigned integer type that’s the same width as pointer. This is why you must be careful never to let a negative length field make its way to a libc routine, such as snprintf(), strcpy(), memcpy(), read(), or strncpy().

This situation occurs fairly often, particularly when signed integers are used for length values and the programmer doesn’t consider the potential for a value less than 0. In this case, all values less than 0 have their value changed to a high positive number when they are converted to an unsigned type. Malicious users can often specify negative integers through various program interfaces and undermine an application’s logic. This type of bug happens commonly when a maximum length check is performed on a user-supplied integer, but no check is made to see whether the integer is negative, as in Listing 6-7.

Listing 6-7
Signed Comparison Vulnerability Example

```c
int read_user_data(int sockfd)
{
    int length, sockfd, n;
    char buffer[1024];

    length = get_user_length(sockfd);

    if(length > 1024){
        error("illegal input, not enough room in buffer\n");
        return -1;
    }

    if(read(sockfd, buffer, length) < 0){
        error("read: %m");
        return -1;
    }

    return 0;
}
```

In Listing 6-7, assume that the get_user_length() function reads a 32-bit integer from the network. If the length the user supplies is negative, the length check can be evaded, and the application can be compromised. A negative length is converted to a size_t type for the call to read(), which as you know, turns into a large unsigned value. A code reviewer should always consider the implications of negative values in signed types and see whether unexpected results can be produced that could lead to security exposures. In this case, a buffer overflow can be triggered because of the erroneous length check; consequently, the oversight is quite serious.
Auditing Tip: Signed/Unsigned Conversions

You want to look for situations in which a function takes a `size_t` or unsigned int length parameter, and the programmer passes in a signed integer that can be influenced by users. Good functions to look for include `read()`, `recvfrom()`, `memcpy()`, `memset()`, `bcopy()`, `snprintf()`, `strncat()`, `strncpy()`, and `malloc()`. If users can coerce the program into passing in a negative value, the function interprets it as a large value, which could lead to an exploitable condition.

Also, look for places where length parameters are read from the network directly or are specified by users via some input mechanism. If the length is interpreted as a signed variable in parts of the code, you should evaluate the impact of a user supplying a negative value.

As you review functions in an application, it’s a good idea to note the data types of each function’s arguments in your function audit log. This way, every time you audit a subsequent call to that function, you can simply compare the types and examine the type conversion tables in this chapter’s “Type Conversions” section to predict exactly what’s going to happen and the implications of that conversion. You learn more about analyzing functions and keeping logs of function prototypes and behavior in Chapter 7, “Program Building Blocks.”

Sign Extension

Sign extension occurs when a smaller signed integer type is converted to a larger type, and the machine propagates the sign bit of the smaller type through the unused bits of the larger type. The intent of sign extension is that the conversion is value-preserving when going from a smaller signed type to a larger signed type.

As you know, sign extension can occur in several ways. First, if a simple conversion is made from a small signed type to a larger type, with a typecast, assignment, or function call, sign extension occurs. You also know that sign extension occurs if a signed type smaller than an integer is promoted via the integer promotions. Sign extension could also occur as a result of the usual arithmetic conversions applied after integer promotions because a signed integer type could be promoted to a larger type, such as long long.

Sign extension is a natural part of the language, and it’s necessary for value-preserving promotions of integers. So why is it mentioned as a security issue? There are two reasons:
In certain cases, sign extension is a value-changing conversion that has an unexpected result.

Programmers consistently forget that the char and short types they use are signed!

To examine the first reason, if you recall from the conversion section, one of the more interesting findings was that sign extension is performed if a smaller signed type is converted into a larger unsigned type. Say a programmer does something like this:

```c
char len;
len=get_len_field();
snprintf(dst, len, "%s", src);
```

This code has disaster written all over it. If the result of `get_len_field()` is such that `len` has a value less than 0, that negative value is passed as the length argument to `snprintf()`. Say the programmer tries to fix this error and does the following:

```c
char len;
len=get_len_field();
snprintf(dst, (unsigned int)len, "%s", src);
```

This solution sort of makes sense. An unsigned integer can’t be negative, right? Unfortunately, sign extension occurs during the conversion from char to unsigned int, so the attempt to get rid of characters less than 0 backfired. If `len` happens to be below 0, `(unsigned int)len` ends up with a large value.

This example might seem somewhat arbitrary, but it’s similar to an actual bug the authors recently discovered in a client’s code. The moral of the story is that you should always remember sign extension is applied when converting from a smaller signed type to a larger unsigned type.

Now for the second reason—programmers consistently forget that the char and short types they use are signed. This statement rings quite true, especially in network code that deals with signed integer lengths or code that processes binary or text data one character at a time. Take a look at a real-world vulnerability in the DNS packet-parsing code of l0pht’s antisniff tool (http://packetstormsecurity.org/sniffers/antisniff/). It’s an excellent bug for demonstrating some vulnerabilities that have been discussed. A buffer overflow was first discovered in the software involving the improper use of `strcat()`, and after that vulnerability was patched, researchers from TESO discovered that it was still vulnerable because of a sign-extension issue. The fix for the sign-extension issue wasn’t correct, and yet another
vulnerability was published. The following examples take you through the timeline of this vulnerability.

Listing 6-8 contains the slightly edited vulnerable code from version 1 of the antisniff research release, in the raw_watchdns.c file in the watch_dns_ptr() function.

Listing 6-8

Antisniff v1.0 Vulnerability

```c
char *indx;
int count;
char nameStr[MAX_LEN]; //256
...
memset(nameStr, '\0', sizeof(nameStr));
...
indx = (char *)(pkt + rr_offset);
count = (char)*indx;
while (count){
    (char *)indx++;
    strncat(nameStr, (char *)indx, count);
    indx += count;
    count = (char)*indx;
    strncat(nameStr, ".",
            sizeof(nameStr) – strlen(nameStr));
}
nameStr[strlen(nameStr)-1] = '\0';
```

Before you can understand this code, you need a bit of background. The purpose of the watch_dns_ptr() function is to extract the domain name from the packet and copy it into the nameStr string. The DNS domain names in DNS packets sort of resemble Pascal strings. Each label in the domain name is prefixed by a byte containing its length. The domain name ends when you reach a label of size 0. (The DNS compression scheme isn’t relevant to this vulnerability.) Figure 6-8 shows what a DNS domain name looks like in a packet. There are three labels—test, jim, and com—and a 0-length label specifying the end of the name.

test.jim.com

![Figure 6-8 Sample DNS domain name](image)
The code starts by reading the first length byte from the packet and storing it in the integer count. This length byte is a signed character stored in an integer, so you should be able to put any value you like between -128 and 127 in count. Keep this in mind for later.

The while() loop keeps reading in labels and calling strcat() on them to the nameStr string. The first vulnerability that was published is no length check in this loop. If you just provide a long enough domain name in the packet, it could write past the bounds of nameStr[]. Listing 6-9 shows how this issue was fixed in version 1.1 of the research version.

**Listing 6-9**

*Antisniff v1.1 Vulnerability*

```c
char *indx;
int count;
char nameStr[MAX_LEN]; //256
...
memset(nameStr, '\0', sizeof(nameStr));
...
indx = (char *)(pkt + rr_offset);
count = (char)*indx;

while (count){
    if (strlen(nameStr) + count < ( MAX_LEN - 1) ){
        (char *)indx++;
        strcat(nameStr, (char *)indx, count);
        indx += count;
        count = (char)*indx;
        strcat(nameStr, ".",
                sizeof(nameStr) - strlen(nameStr));
    } else {
        fprintf(stderr, "Alert! Someone is attempting "
                "to send LONG DNS packets\n");
        count = 0;
    }
}
nameStr[strlen(nameStr)-1] = '\0';
```

The code is basically the same, but length checks have been added to try to prevent the buffer from being overflowed. At the top of the loop, the program checks to make sure there’s enough space in the buffer for count bytes before it does the string concatenation. Now examine this code with sign-extension vulnerabilities in mind. You know that count can be any value between -128 and 127, so what happens if you give a negative value for count? Look at the length check:

```c
if (strlen(nameStr) + count < ( MAX_LEN - 1) ){
```
You know that \texttt{strlen(nameStr)} is going to return a \texttt{size_t}, which is effectively an unsigned int on a 32-bit system, and you know that \texttt{count} is an integer below 0. Say you’ve been through the loop once, and \texttt{strlen(nameStr)} is 5, and \texttt{count} is -1. For the addition, \texttt{count} is converted to an unsigned integer, and you have \((5 + 4,294,967,295)\). This addition can easily cause a numeric overflow so that you end up with a small value, such as 4; 4 is less than \((\text{MAX\_LEN} - 1)\), which is 256. So far, so good. Next, you see that \texttt{count} (which you set to -1), is passed in as the length argument to \texttt{strncat()}. The \texttt{strncat()} function takes a \texttt{size_t}, so it interprets that as \(4,294,967,295\). Therefore, you win again because you can essentially append as much information as you want to the \texttt{nameStr} string.

Listing 6-10 shows how this vulnerability was fixed in version 1.1.1 of the research release.

**Listing 6-10**

\textit{Antisniff v1.1.1 Vulnerability}

```c
char *indx;
int count;
char nameStr[MAX_LEN]; //256
...
memset(nameStr, '\0', sizeof(nameStr));
...
indx = (char *)(pkt + rr_offset);
count = (char)*indx;

while (count){
    /* typecast the strlen so we aren't dependent on 
       the call to be properly setting to unsigned. */
    if ((unsigned int)strlen(nameStr) +
        (unsigned int)count < ( MAX_LEN - 1 )){
        (char *)indx++;
        strncat(nameStr, (char *)indx, count);
        indx += count;
        count = (char)*indx;
        strncat(nameStr, ".", 
            sizeof(nameStr) - strlen(nameStr));
    } else {
        fprintf(stderr, "Alert! Someone is attempting 
            "to send LONG DNS packets\n");
        count = 0;
    }
}

nameStr[strlen(nameStr)-1] = '\0';
```

This solution is basically the same code, except some typecasts have been added to the length check. Take a closer look:
if ((unsigned int)strlen(nameStr) +
     (unsigned int)count < (MAX_LEN - 1)) {

The result of `strlen()` is typecast to an unsigned int, which is superfluous because it’s already a size_t. Then `count` is typecast to an unsigned int. This is also superfluous, as it’s normally converted to an unsigned integer type by the addition operator. In essence, nothing has changed. You can still send a negative label length and bypass the length check! Listing 6-11 shows how this problem was fixed in version 1.1.2.

Listing 6-11
Antisniff v1.1.2 Vulnerability

```c
unsigned char *indx;
unsigned int count;
unsigned char nameStr[MAX_LEN]; //256
...
memset(nameStr, '\0', sizeof(nameStr));
...
indx = (char *)(pkt + rr_offset);
count = (char)*indx;
while (count){
    if (strlen(nameStr) + count < (MAX_LEN - 1)) {
        indx++;
        strncat(nameStr, indx, count);
        indx += count;
        count = *indx;
        strncat(nameStr, ".",
                 sizeof(nameStr) - strlen(nameStr));
    } else {
        fprintf(stderr, "Alert! Someone is attempting "
                 "to send LONG DNS packets\n");
        count = 0;
    }
}
nameStr[strlen(nameStr)-1] = '\0';
```

The developers have changed `count`, `nameStr`, and `indx` to be unsigned and changed back to the previous version’s length check. So the sign extension you were taking advantage of now appears to be gone because the character pointer, `indx`, is now an unsigned type. However, take a closer look at this line:

```
count = (char)*indx;
```

This code line dereferences `indx`, which is an unsigned char pointer. This gives you an unsigned character, which is then explicitly converted into a signed char. You know the bit pattern won’t change, so you’re back to something with a range of
-128 to 127. It’s assigned to an unsigned int, but you know that converting from a smaller signed type to a larger unsigned type causes sign extension. So, because of the typecast to (char), you still can get a maliciously large count into the loop, but only for the first label. Now look at that length check with this in mind:

```c
if (strlen(nameStr) + count < (MAX_LEN - 1)) {

    Unfortunately, strlen(nameStr) is 0 when you enter the loop for the first time. So the rather large value of count won’t be less than (MAX_LEN - 1), and you get caught and kicked out of the loop. Close, but no cigar. Amusingly, if you do get kicked out on your first trip into the loop, the program does the following:

    nameStr[strlen(nameStr)-1] = '\0';

    Because strlen(nameStr) is 0, that means it writes a 0 at 1 byte behind the buffer, at nameStr[-1]. Now that you’ve seen the evolution of the fix from the vantage point of 20-20 hindsight, take a look at Listing 6-12, which is an example based on a short integer data type.

Listing 6-12
Sign Extension Vulnerability Example
unsigned short read_length(int sockfd)
{
    unsigned short len;

    if(full_read(sockfd, (void *)&len, 2) != 2)
        die("could not read length!\n");

    return ntohs(len);
}

int read_packet(int sockfd)
{
    struct header hdr;
    short length;
    char *buffer;

    length = read_length(sockfd);

    if(length > 1024){
        error("read_packet: length too large: %d\n", length);
        return -1;
    }

    buffer = (char *)malloc(length+1);
```
if((n = read(sockfd, buffer, length) < 0){
  error("read: %m");
  free(buffer);
  return -1;
}

buffer[n] = '\0';

return 0;
}

Several concepts you’ve explored in this chapter are in effect here. First, the result of the read_length() function, an unsigned short int, is converted into a signed short int and stored in length. In the following length check, both sides of the comparison are promoted to integers. If length is a negative number, it passes the check that tests whether it’s greater than 1024. The next line adds 1 to length and passes it as the first argument to malloc(). The length parameter is again sign-extended because it’s promoted to an integer for the addition. Therefore, if the specified length is 0xFFFF, it’s sign-extended to 0xFFFFFFFF. The addition of this value plus 1 wraps around to 0, and malloc(0) potentially returns a small chunk of memory. Finally, the call to read() causes the third argument, the length parameter, to be converted directly from a signed short int to a size_t. Sign extension occurs because it’s a case of a smaller signed type being converted to a larger unsigned type. Therefore, the call to read allows you to read a large number of bytes into the buffer, resulting in a potential buffer overflow.

Another quintessential example of a place where programmers forget whether small types are signed occurs with use of the ctype libc functions. Consider the toupper() function, which has the following prototype:

int toupper(int c);

The toupper() function works on most libc implementations by searching for the correct answer in a lookup table. Several libcs don’t handle a negative argument correctly and index behind the table in memory. The following definition of toupper() isn’t uncommon:

int toupper(int c)
{
  return _toupper_tab[c];
}

Say you do something like this:

void upperize(char *str)
If you have a libc implementation that doesn’t have a robust `toupper()` function, you could end up making some strange changes to your string. If one of the characters is -1, it gets converted to an integer with the value -1, and the `toupper()` function indexes behind its table in memory.

Take a look at a final real-world example of programmers not considering sign extension. Listing 6-13 is a Sendmail vulnerability that security researcher Michael Zalewski discovered (www.cert.org/advisories/CA-2003-12.html). It’s from Sendmail version 8.12.3 in the `prescan()` function, which is primarily responsible for parsing e-mail addresses into tokens (from `sendmail/parseaddr.c`). The code has been edited for brevity.

**Listing 6-13**

*Prescan Sign Extension Vulnerability in Sendmail*

```c
register char *p;
register char *q;
register int c;
...
p = addr;

for (;;) {
    /* store away any old lookahead character */
    if (c != NOCHAR && !bslashmode) {
        /* see if there is room */
        if (q >= &pvpbuf[pvpbsize - 5]) {
            usrrr("553 5.1.1 Address too long");
            if (strlen(addr) > MAXNAME)
                addr[MAXNAME] = '\0';
        returnnull:
            if (delimptr != NULL)
                *delimptr = p;
            CurEnv->e_to = saveto;
            return NULL;
        }
    }
```
The `NOCHAR` constant is defined as -1 and is meant to signify certain error conditions when characters are being processed. The `p` variable is processing a user-supplied address and exits the loop shown when a complete token has been read. There’s a length check in the loop; however, it’s examined only when two conditions are true: when `c` is not `NOCHAR` (that is, `c != -1`) and `bslashmode` is false. The problem is this line:

\[ c = *p++; \]

Because of the sign extension of the character that `p` points to, users can specify the char `0xFF` and have it extended to `0xFFFFFFFF`, which is `NOCHAR`. If users supply a repeating pattern of `0x2F` (backslash character) followed by `0xFF`, the loop can run continuously without ever performing the length check at the top. This causes backslashes to be written continually into the destination buffer without checking whether enough
room is left. Therefore, because of the character being sign-extended when stored in
the variable c, an unexpected code path is triggered that results in a buffer overflow.

This vulnerability also reinforces another principle stated at the beginning of
this chapter. Implicit actions performed by the compiler are subtle, and when
reviewing source code, you need to examine the implications of type conversions
and anticipate how the program will deal with unexpected values (in this case, the
NOCHAR value, which users can specify because of the sign extension).

Sign extension seems as though it should be ubiquitous and mostly harmless in C
code. However, programmers rarely intend for their smaller data types to be sign-
extended when they are converted, and the presence of sign extension often indicates
a bug. Sign extension is somewhat difficult to locate in C, but it shows up well in
assembly code as the move instruction. Try to practice searching through assembly
for sign-extension conversions and then relating them back to the source code, which
is a useful technique.

As a brief demonstration, compare Listings 6-14 and 6-15.

Listing 6-14
Sign-Extension Example

```c
unsigned int l;
char c=5;
l=c;
```

Listing 6-15
Zero-Extension Example

```c
unsigned int l;
unsigned char c=5;
l=c;
```

Assuming the implementation calls for signed characters, you know that sign
extension will occur in Listing 6-14 but not in Listing 6-15. Compare the generated
assembly code, reproduced in Table 6-8.

<table>
<thead>
<tr>
<th>Table 6-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign Extension Versus Zero Extension in Assembly Code</td>
</tr>
<tr>
<td>Listing 6-14: Sign Extension</td>
</tr>
<tr>
<td>mov [ebp+var_5], 5</td>
</tr>
<tr>
<td>movsx eax, [ebp+var_5]</td>
</tr>
<tr>
<td>mov [ebp+var_4], eax</td>
</tr>
</tbody>
</table>
You can see that in the sign-extension example, the `movsx` instruction is used. In the zero-extension example, the compiler first clears the register with `xor eax, eax` and then moves the character byte into that register.

### Auditing Tip: Sign Extension

When looking for vulnerabilities related to sign extensions, you should focus on code that handles signed character values or pointers or signed short integer values or pointers. Typically, you can find them in string-handling code and network code that decodes packets with length elements. In general, you want to look for code that takes a character or short integer and uses it in a context that causes it to be converted to an integer. Remember that if you see a signed character or signed short converted to an unsigned integer, sign extension still occurs.

As mentioned previously, one effective way to find sign-extension vulnerabilities is to search the assembly code of the application binary for the `movsx` instruction. This technique can often help you cut through multiple layers of typedefs, macros, and type conversions when searching for potentially vulnerable locations in code.

### Truncation

Truncation occurs when a larger type is converted into a smaller type. Note that the usual arithmetic conversions and the integral promotions never really call for a large type to be converted to a smaller type. Therefore, truncation can occur only as the result of an assignment, a typecast, or a function call involving a prototype. Here’s a simple example of truncation:

```c
int g = 0x12345678;
short int h;

h = g;
```

When g is assigned to h, the top 16 bits of the value are truncated, and h has a value of 0x5678. So if this data loss occurs in a situation the programmer didn’t expect, it could certainly lead to security failures. Listing 6-16 is loosely based on a historic vulnerability in Network File System (NFS) that involves integer truncation.
Listing 6-16
Truncation Vulnerability Example in NFS

```c
void assume_privs(unsigned short uid)
{
    seteuid(uid);
    setuid(uid);
}

int become_user(int uid)
{
    if (uid == 0)
        die("root isn't allowed");

    assume_privs(uid);
}
```

To be fair, this vulnerability is mostly known of anecdotally, and its existence hasn’t been verified through source code. NFS forbids users from mounting a disk remotely with root privileges. Eventually, attackers figured out that they could specify a UID of 65536, which would pass the security checks that prevent root access. However, this UID would get assigned to an unsigned short integer and be truncated to a value of 0. Therefore, attackers could assume root’s identity of UID 0 and bypass the protection.

Take a look at one more synthetic vulnerability in Listing 6-17 before looking at a real-world truncation issue.

Listing 6-17
Truncation Vulnerability Example

```c
unsigned short int f;
char mybuf[1024];
char *userstr=getuserstr();

f=strlen(userstr);
if (f > sizeof(mybuf)-5)
    die("string too long!");
strcpy(mybuf, userstr);
```

The result of the `strlen()` function, a `size_t`, is converted to an unsigned short. If a string is 66,000 characters long, truncation would occur and `f` would have the value 464. Therefore, the length check protecting `strcpy()` would be circumvented, and a buffer overflow would occur.

A show-stopping bug in most SSH daemons was caused by integer truncation. Ironically, the vulnerable code was in a function designed to address another security hole, the SSH insertion attack identified by CORE-SDI. Details on that attack are available at www1.corest.com/files/files/11/CRC32.pdf.
The essence of the attack is that attackers can use a clever known plain-text attack against the block cipher to insert small amounts of data of their choosing into the SSH stream. Normally, this attack would be prevented by message integrity checks, but SSH used CRC32, and the researchers at CORE-SDI figured out how to circumvent it in the context of the SSH protocol.

The responsibility of the function containing the truncation vulnerability is to determine whether an insertion attack is occurring. One property of these insertion attacks is a long sequence of similar bytes at the end of the packet, with the purpose of manipulating the CRC32 value so that it’s correct. The defense that was engineered was to search for repeated blocks in the packet, and then do the CRC32 calculation up to the point of repeat to determine whether any manipulation was occurring. This method was easy for small packets, but it could have a performance impact on large sets of data. So, presumably to address the performance impact, a hashing scheme was used.

The function you’re about to look at has two separate code paths. If the packet is below a certain size, it performs a direct analysis of the data. If it’s above that size, it uses a hash table to make the analysis more efficient. It isn’t necessary to understand the function to appreciate the vulnerability. If you’re curious, however, you’ll see that the simpler case for the smaller packets has roughly the algorithm described in Listing 6-18.

**Listing 6-18**

*Detect_attack Small Packet Algorithm in SSH*

```plaintext
for c = each 8 byte block of the packet
    if c is equal to the initialization vector block
        check c for the attack.
        If the check succeeds, return DETECTED.
        If the check fails, you aren’t under attack so return OK.
    for d = each 8 byte block of the packet before c
        if d is equal to c, check c for the attack.
        If the check succeeds, return DETECTED.
        If the check fails, break out of the d loop.
    next d
next c
```

The code goes through each 8-byte block of the packet, and if it sees an identical block in the packet before the current one, it does a check to see whether an attack is underway.

The hash-table-based path through the code is a little more complex. It has the same general algorithm, but instead of comparing a bunch of 8-byte blocks with each other, it takes a 32 bit hash of each block and compares them. The hash table is indexed by the 32-bit hash of the 8-byte block, modulo the hash table size, and the bucket contains the position of the block that last hashed to that bucket.
The truncation problem happened in the construction and management of the hash table. Listing 6-19 contains the beginning of the code.

**Listing 6-19**

_Detect_attack Truncation Vulnerability in SSH_

/* Detect a crc32 compensation attack on a packet */
int
detect_attack(unsigned char *buf, u_int32_t len,
               unsigned char *IV)
{
    static u_int16_t *h = (u_int16_t *) NULL;
    static u_int16_t n = HASH_MINSIZE / HASH_ENTRYSIZE;
    register u_int32_t i, j;
    u_int32_t l;
    register unsigned char *c;
    unsigned char *d;
    if (len > (SSH_MAXBLOCKS * SSH_BLOCKSIZE) ||
        len % SSH_BLOCKSIZE != 0) {
        fatal("detect_attack: bad length %d", len);
    }

    First, the code checks whether the packet is overly long or isn’t a multiple of 8 bytes. SSH_MAXBLOCKS is 32,768 and BLOCKSIZE is 8, so the packet can be as large as 262,144 bytes. In the following code, \( n \) starts out as \( \text{HASH\_MINSIZE} / \text{HASH\_ENTRYSIZE} \), which is 8,192 / 2, or 4,096, and its purpose is to hold the number of entries in the hash table:

    for (l = n; l < HASH_FACTOR(len / SSH_BLOCKSIZE); l = l << 2) {

    The starting size of the hash table is 8,192 elements. This loop attempts to determine a good size for the hash table. It starts off with a guess of \( n \), which is the current size, and it checks to see whether it’s big enough for the packet. If it’s not, it quadruples \( l \) by shifting it left twice. It decides whether the hash table is big enough by making sure there are \( 3/2 \) the number of hash table entries as there are 8-byte blocks in the packet. \( \text{HASH\_FACTOR} \) is defined as \( ((x)*3/2) \). The following code is the interesting part:

    if (h == NULL) {
        debug("Installing crc compensation "
               "attack detector.");
        n = l;
        h = (u_int16_t *) xmalloc(n * HASH_ENTRYSIZE);
    } else {

262
if \( l > n \) {  
    n = l;  
    h = (\text{u_int16_t} *)\text{xrealloc}(h, n * \text{HASH_ENTRYSIZE});  
}  

If \( h \) is NULL, that means it’s your first time through this function and you need to allocate space for a new hash table. If you remember, \( l \) is the value calculated as the right size for the hash table, and \( n \) contains the number of entries in the hash table. If \( h \) isn’t NULL, the hash table has already been allocated. However, if the hash table isn’t currently big enough to agree with the newly calculated \( l \), you go ahead and reallocate it.

You’ve looked at enough code so far to see the problem: \( n \) is an unsigned short int. If you send a packet that’s big enough, \( l \), an unsigned int, could end up with a value larger than 65,535, and when the assignment of \( l \) to \( n \) occurs, truncation could result. For example, assume you send a packet that’s 262,144 bytes. It passes the first check, and then in the loop, \( l \) changes like so:

Iteration 1: \( l = 4096 \)  \( l < 49152 \)  \( l<<=4 \)
Iteration 2: \( l = 16384 \)  \( l < 49152 \)  \( l<<=4 \)
Iteration 3: \( l = 65536 \)  \( l >= 49152 \)

When \( l \), with a value of 65,536, is assigned to \( n \), the top 16 bits are truncated, and \( n \) ends up with a value of 0. On several modern OSs, a \text{malloc()} of 0 results in a valid pointer to a small object being returned, and the rest of the function’s behavior is extremely suspect.

The next part of the function is the code that does the direct analysis, and because it doesn’t use the hash table, it isn’t of immediate interest:

```c
if (len <= HASH_MINBLOCKS) {  
    for (c = buf; c < buf + len; c += SSH_BLOCKSIZE) {  
        if (IV && (!CMP(c, IV))) {  
            if ((check_cbc(c, buf, len, IV)))  
                return (DEATTACK_DETECTED);  
            else  
                break;  
        }  
    for (d = buf; d < c; d += SSH_BLOCKSIZE) {  
        if (!CMP(c, d)) {  
            if ((check_cbc(c, buf, len, IV)))
```
return (DEATTACK_DETECTED);
else
break;
}
}
return (DEATTACK_OK);
}

Next is the code that performs the hash-based detection routine. In the following code, keep in mind that n is going to be 0 and h is going to point to a small but valid object in the heap. With these values, it's possible to do some interesting things to the process's memory:

memset(h, HASH_UNUSEDCHAR, n * HASH_ENTRYSIZE);

if (IV)
    h[HASH(IV) & (n - 1)] = HASH_IV;
for (c = buf, j = 0; c < (buf + len); c += SSH_BLOCKSIZE, j++) {
    for (i = HASH(c) & (n - 1); h[i] != HASH_UNUSED;
        i = (i + 1) & (n - 1)) {
        if (h[i] == HASH_IV) {
            if (!CMP(c, IV)) {
                if (check_crc(c, buf, len, IV))
                    return (DEATTACK_DETECTED);
                else
                    break;
            } else if (!CMP(c, buf + h[i] * SSH_BLOCKSIZE)) {
                if (check_crc(c, buf, len, IV))
                    return (DEATTACK_DETECTED);
                else
                    break;
        } else if (!CMP(c, buf + h[i] * SSH_BLOCKSIZE)) {
            if (check_crc(c, buf, len, IV))
                return (DEATTACK_DETECTED);
        else
            break;
    }
    h[i] = j;
}
If you don’t see an immediate way to attack this loop, don’t worry. (You are in good company, and also some critical macro definitions are missing.) This bug is extremely subtle, and the exploits for it are complex and clever. In fact, this vulnerability is unique from many perspectives. It reinforces the notion that secure programming is difficult, and everyone can make mistakes, as CORE-SDI is easily one of the world’s most technically competent security companies. It also demonstrates that sometimes a simple black box test can uncover bugs that would be hard to find with a source audit; the discoverer, Michael Zalewski, located this vulnerability in a stunningly straightforward fashion (ssh -l long_user_name). Finally, it highlights a notable case in which writing an exploit can be more difficult than finding its root vulnerability.

Auditing Tip: Truncation
Truncation-related vulnerabilities are typically found where integer values are assigned to smaller data types, such as short integers or characters. To find truncation issues, look for locations where these shorter data types are used to track length values or to hold the result of a calculation. A good place to look for potential variables is in structure definitions, especially in network-oriented code.

Programmers often use a short or character data type just because the expected range of values for a variable maps to that data type nicely. Using these data types can often lead to unanticipated truncations, however.

Comparisons
You’ve already seen examples of signed comparisons against negative numbers in length checks and how they can lead to security exposures. Another potentially hazardous situation is comparing two integers that have different types. As you’ve learned, when a comparison is made, the compiler first performs integer promotions on the operands and then follows the usual arithmetic conversions on the operands so that a comparison can be made on compatible types. Because these promotions and conversions might result in value changes (because of sign change), the comparison might not be operating exactly as the programmer intended. Attackers can take advantage of these conversions to circumvent security checks and often compromise an application.

To see how comparisons can go wrong, take a look at Listing 6-20. This code reads a short integer from the network, which specifies the length of an incoming packet. The first half of the length check compares (length - sizeof(short)) with
0 to make sure the specified length isn’t less than sizeof(short). If it is, it could wrap around to a large integer when sizeof(short) is subtracted from it later in the read() statement.

Listing 6-20
Comparison Vulnerability Example
#define MAX_SIZE 1024

int read_packet(int sockfd)
{
    short length;
    char buf[MAX_SIZE];

    length = network_get_short(sockfd);

    if(length - sizeof(short) <= 0 || length > MAX_SIZE){
        error("bad length supplied\n");
        return -1;
    }

    if(read(sockfd, buf, length - sizeof(short)) < 0){
        error("read: %m\n");
        return -1;
    }

    return 0;
}

The first check is actually incorrect. Note that the result type of the sizeof operator is a size_t, which is an unsigned integer type. So for the subtraction of (length - sizeof(short)), length is first promoted to a signed int as part of the integer promotions, and then converted to an unsigned integer type as part of the usual arithmetic conversions. The resulting type of the subtraction operation is an unsigned integer type. Consequently, the result of the subtraction can never be less than 0, and the check is effectively inoperative. Providing a value of 1 for length evades the very condition that the length check in the first half of the if statement is trying to protect against and triggers an integer underflow in the call to read().

More than one value can be supplied to evade both checks and trigger a buffer overflow. If length is a negative number, such as 0xFFFF, the first check still passes because the result type of the subtraction is always unsigned. The second check also passes (length > MAX_SIZE) because length is promoted to a signed int for the comparison and retains its negative value, which is less than MAX_SIZE (1024). This result demonstrates that the length variable is treated as unsigned in one case and signed in another case because of the other operands used in the comparison.
When dealing with data types smaller than int, integer promotions cause narrow values to become signed integers. This is a value-preserving promotion and not much of a problem in itself. However, sometimes comparisons can be promoted to a signed type unintentionally. Listing 6-21 illustrates this problem.

**Listing 6-21**  
*Signed Comparison Vulnerability*

```c
int read_data(int sockfd)  
{  
    char buf[1024];  
    unsigned short max = sizeof(buf);  
    short length;  
    
    length = get_network_short(sockfd);  
    
    if(length > max){  
        error("bad length: %d\n", length);  
        return -1;  
    }  
    
    if(read(sockfd, buf, length) < 0){  
        error("read: %m");  
        return -1;  
    }  
    
    ... process data ...  
    
    return 0;  
}
```

Listing 6-21 illustrates why you must be aware of the resulting data type used in a comparison. Both the `max` and `length` variables are short integers and, therefore, go through integer conversions; both get promoted to signed integers. This means any negative value supplied in `length` evades the length check against `max`. Because of data type conversions performed in a comparison, not only can sanity checks be evaded, but the entire comparison could be rendered useless because it’s checking for an impossible condition. Consider Listing 6-22.

**Listing 6-22**  
*Unsigned Comparison Vulnerability*

```c
int get_int(char *data)  
{  
    unsigned int n = atoi(data);  
    if(n < 0 || n > 1024)  
        return -1;  
}
```
Listing 6-22 checks the variable `n` to make sure it falls within the range of 0 to 1024. Because the variable `n` is unsigned, however, the check for less than 0 is impossible. An unsigned integer can never be less than 0 because every value that can be represented is positive. The potential vulnerability is somewhat subtle; if attackers provide an invalid integer as `argv[1]`, `get_int()` returns a -1, which is converted to an unsigned long when assigned to `n`. Therefore, it would become a large value and end up causing `memset()` to crash the program.

Compilers can detect conditions that will never be true and issue a warning if certain flags are passed to it. See what happens when the preceding code is compiled with GCC:

```
[root@doppelganger root]# gcc -Wall -o example example.c
[root@doppelganger root]# gcc -W -o example example.c
example.c: In function 'get_int':
example.c:10: warning: comparison of unsigned expression < 0 is always false
example.c: In function 'main':
example.c:25: warning: comparison of unsigned expression < 0 is always false
[root@doppelganger root]#
```
Notice that the `-Wall` flag doesn’t warn about this type of error as most developers would expect. To generate a warning for this type of bug, the `-W` flag must be used. If the code `if(n < 0)` is changed to `if(n <= 0)`, a warning isn’t generated because the condition is no longer impossible. Now take a look at a real-world example of a similar mistake. Listing 6-23 is taken from the PHP Apache module (4.3.4) when reading POST data.

**Listing 6-23**

*Signed Comparison Example in PHP*

```c
/* {{{ sapi_apache_read_post */
static int sapi_apache_read_post(char *buffer,
                                 int count_bytes TSRMLS_DC)
{
    uint total_read_bytes=0, read_bytes;
    request_rec *r = (request_rec *) SG(server_context);
    void (*handler)(int);

    /*
     * This handles the situation where the browser sends a
     * Expect: 100-continue header and needs to receive
     * confirmation from the server on whether or not it
     * can send the rest of the request. RFC 2616
     *
     *
     */
    if (!SG(read_post_bytes) && !ap_should_client_block(r)) {
        return total_read_bytes;
    }

    handler = signal(SIGPIPE, SIG_IGN);
    while (total_read_bytes<count_bytes) {
        /* start timeout timer */
        hard_timeout("Read POST information", r);
        read_bytes = get_client_block(r,
                                       buffer + total_read_bytes,
                                       count_bytes - total_read_bytes);
        reset_timeout(r);
        if (read_bytes<=0) {
            break;
        }
        total_read_bytes += read_bytes;
    }
    signal(SIGPIPE, handler);
    return total_read_bytes;
}
```

Type Conversion Vulnerabilities
The return value from `get_client_block()` is stored in the `read_bytes` variable and then compared to make sure a negative number wasn’t returned. Because `read_bytes` is unsigned, this check doesn’t detect errors from `get_client_block()` as intended. As it turns out, this bug isn’t immediately exploitable in this function. Can you see why? The loop controlling the loop also has an unsigned comparison, so if `total_read_bytes` is decremented under 0, it underflows and, therefore, takes a value larger than `count_bytes`, thus exiting the loop.

**Auditing Tip**
Reviewing comparisons is essential to auditing C code. Pay particular attention to comparisons that protect allocation, array indexing, and copy operations. The best way to examine these comparisons is to go line by line and carefully study each relevant expression.

In general, you should keep track of each variable and its underlying data type. If you can trace the input to a function back to a source you’re familiar with, you should have a good idea of the possible values each input variable can have.

Proceed through each potentially interesting calculation or comparison, and keep track of potential values of the variables at different points in the function evaluation. You can use a process similar to the one outlined in the previous section on locating integer boundary condition issues.

When you evaluate a comparison, be sure to watch for unsigned integer values that cause their peer operands to be promoted to unsigned integers. `sizeof` and `strlen()` are classic examples of operands that cause this promotion.

Remember to keep an eye out for unsigned variables used in comparisons, like the following:
```
if (uvar < 0) ...
if (uvar <= 0) ...
```

The first form typically causes the compiler to emit a warning, but the second form doesn’t. If you see this pattern, it’s a good indication something is probably wrong with that section of the code. You should do a careful line-by-line analysis of the surrounding functionality.
Operators

Operators can produce unanticipated results. As you have seen, unsanitized operands used in simple arithmetic operations can potentially open security holes in applications. These exposures are generally the result of crossing over boundary conditions that affect the meaning of the result. In addition, each operator has associated type promotions that are performed on each of its operands implicitly which could produce some unexpected results. Because producing unexpected results is the essence of vulnerability discovery, it’s important to know how these results might be produced and what exceptional conditions could occur. The following sections highlight these exceptional conditions and explain some common misuses of operators that could lead to potential vulnerabilities.

The sizeof Operator

The first operator worth mentioning is sizeof. It’s used regularly for buffer allocations, size comparisons, and size parameters to length-oriented functions. The sizeof operator is susceptible to misuse in certain circumstances that could lead to subtle vulnerabilities in otherwise solid-looking code.

One of the most common mistakes with sizeof is accidentally using it on a pointer instead of its target. Listing 6-24 shows an example of this error.

Listing 6-24

Sizeof Misuse Vulnerability Example

```c
char *read_username(int sockfd)
{
    char *buffer, *style, userstring[1024];
    int i;
    buffer = (char *)malloc(1024);
    if(!buffer){
        error("buffer allocation failed: %m");
        return NULL;
    }
    if(read(sockfd, userstring, sizeof(userstring)-1) <= 0){
        free(buffer);
        error("read failure: %m");
        return NULL;
    }
    userstring[sizeof(userstring)-1] = '\0';
    style = strchr(userstring, ':');
```
if(style)
    *style++ = '\0';

sprintf(buffer, "username=\%32s", userstring);

if(style)
    snprintf(buffer, sizeof(buffer)-strlen(buffer)-1,
            ", style=%s\n", style);

return buffer;
}

In this code, some user data is read in from the network and copied into the allocated buffer. However, sizeof is used incorrectly on buffer. The intention is for sizeof(buffer) to return 1024, but because it’s used on a character pointer type, it returns only 4! This results in an integer underflow condition in the size parameter to snprintf() when a style value is present; consequently, an arbitrary amount of data can be written to the memory pointed to by the buffer variable. This error is quite easy to make and often isn’t obvious when reading code, so pay careful attention to the types of variables passed to the sizeof operator. They occur most frequently in length arguments, as in the preceding example, but they can also occur occasionally when calculating lengths for allocating space. The reason this type of bug is somewhat rare is that the misallocation would likely cause the program to crash and, therefore, get caught before release in many applications (unless it’s in a rarely traversed code path).

sizeof() also plays an integral role in signed and unsigned comparison bugs (explored in the “Comparison” section previously in this chapter) and structure padding issues (explored in “Structure Padding” later in this chapter).

**Auditing Tip: sizeof**
Be on the lookout for uses of sizeof in which developers take the size of a pointer to a buffer when they intend to take the size of the buffer. This often happens because of editing mistakes, when a buffer is moved from being within a function to being passed into a function.

Again, look for sizeof in expressions that cause operands to be converted to unsigned values.

**Unexpected Results**
You have explored two primary idiosyncrasies of arithmetic operators: boundary conditions related to the storage of integer types and issues caused by conversions that occur when arithmetic operators are used in expressions. A few other nuances
of C can lead to unanticipated behaviors, specifically nuances related to underlying machine primitives being aware of signed-ness. If a result is expected to fall within a specific range, attackers can sometimes violate those expectations.

Interestingly enough, on twos complement machines, there are only a few operators in C in which the signed-ness of operands can affect the result of the operation. The most important operators in this group are comparisons. In addition to comparisons, only three other C operators have a result that’s sensitive to whether operands are signed: right shift (>>), division (/), and modulus (%). These operators can produce unexpected negative results when they’re used with signed operands because of their underlying machine-level operations being sign-aware. As a code reviewer, you should be on the lookout for misuse of these operators because they can produce results that fall outside the range of expected values and catch developers off-guard.

The right shift operator (>>) is often used in applications in place of the division operator (when dividing by powers of 2). Problems can happen when using this operator with a signed integer as the left operand. When right-shifting a negative value, the sign of the value is preserved by the underlying machine performing a sign-extending arithmetic shift. This sign-preserving right shift is shown in Listing 6-25.

Listing 6-25
Sign-Preserving Right Shift
signed char c = 0x80;
c >>= 4;
1000 0000 – value before right shift
1111 1000 – value after right shift

Listing 6-26 shows how this code might produce an unexpected result that leads to a vulnerability. It’s close to an actual vulnerability found recently in client code.

Listing 6-26
Right Shift Vulnerability Example
int print_high_word(int number)
{
    char buf[sizeof("65535")];
    sprintf(buf, "%u", number >> 16);
    return 0;
}

This function is designed to print a 16-bit unsigned integer (the high 16 bits of the number argument). Because number is signed, the right shift sign-extends number by 16 bits if it’s negative. Therefore, the %u specifier to sprintf() has the capability
of printing a number much larger than sizeof("65535"), the amount of space allocated for the destination buffer, so the result is a buffer overflow. Vulnerable right shifts are good examples of bugs that are difficult to locate in source code yet readily visible in assembly code. In Intel assembly code, a signed, or arithmetic, right shift is performed with the sar mnemonic. A logical, or unsigned, right shift is performed with the shr mnemonic. Therefore, analyzing the assembly code can help you determine whether a right shift is potentially vulnerable to sign extension. Table 6-9 shows signed and unsigned right-shift operations in the assembly code.

Table 6-9

<table>
<thead>
<tr>
<th>Signed Right-Shift Operations</th>
<th>Unsigned Right-Shift Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, [ebp+8]</td>
<td>mov eax, [ebp+8]</td>
</tr>
<tr>
<td>sar eax, 16</td>
<td>shr eax, 16</td>
</tr>
<tr>
<td>push eax</td>
<td>push eax</td>
</tr>
<tr>
<td>push offset string</td>
<td>push offset string</td>
</tr>
<tr>
<td>lea eax, [ebp+var_8]</td>
<td>lea eax, [ebp+var_8]</td>
</tr>
<tr>
<td>push eax</td>
<td>push eax</td>
</tr>
<tr>
<td>call sprintf</td>
<td>call sprintf</td>
</tr>
</tbody>
</table>

Division (/) is another operator that can produce unexpected results because of sign awareness. Whenever one of the operands is negative, the resulting quotient is also negative. Often, applications don’t account for the possibility of negative results when performing division on integers. Listing 6-27 shows how using negative operands could create a vulnerability with division.

Listing 6-27

Division Vulnerability Example

```c
int read_data(int sockfd)
{
    int bitlength;
    char *buffer;

    bitlength = network_get_int(length);

    buffer = (char *)malloc(bitlength / 8 + 1);

    if (buffer == NULL)
        die("no memory");

    if(read(sockfd, buffer, bitlength / 8) < 0){
        error("read error: %m");
    }
```
Listing 6-27 takes a `bitlength` parameter from the network and allocates memory based on it. The `bitlength` is divided by 8 to obtain the number of bytes needed for the data that’s subsequently read from the socket. One is added to the result, presumably to store extra bits in if the supplied `bitlength` isn’t a multiple of 8. If the division can be made to return -1, the addition of 1 produces 0, resulting in a small amount of memory being allocated by `malloc()`. Then the third argument to `read()` would be -1, which would be converted to a `size_t` and interpreted as a large positive value.

Similarly, the modulus operator (%) can produce negative results when dealing with a negative dividend operand. Code auditors should be on the lookout for modulus operations that don’t properly sanitize their dividend operands because they could produce negative results that might create a security exposure. Modulus operators are often used when dealing with fixed-sized arrays (such as hash tables), so a negative result could immediately index before the beginning of the array, as shown in Listing 6-28.

Listing 6-28

*Modulus Vulnerability Example*

```c
#define SESSION_SIZE 1024

struct session {
    struct session *next;
    int session_id;
};

struct header {
    int session_id;
    ...
};

struct session *sessions[SESSION_SIZE];

struct session *session_new(int session_id)
{
    struct session *new1, *tmp;

    new1 = malloc(sizeof(struct session));
    if(!new1)
        die("malloc: %m");

    new1->session_id = session_id;
```
new1->next = NULL;

if(!sessions[session_id%(SESSION_SIZE-1)])
{
    sessions[session_id%(SESSION_SIZE-1)] = new1;
    return new1;
}

for(tmp = sessions[session_id%(SESSION_SIZE-1)]; tmp->next;
    tmp = tmp->next);

tmp->next = new1;

return new1;

int read_packet(int sockfd)
{
    struct session *session;
    struct header hdr;

    if(full_read(sockfd, (void *)&hdr, sizeof(hdr)) !=
        sizeof(hdr))
    {
        error("read: %m");
        return -1;
    }

    if((session = session_find(hdr.session_id)) == NULL)
    {
        session = session_new(hdr.sessionid);
        return 0;
    }

    ... validate packet with session ...

    return 0;
}

As you can see, a header is read from the network, and session information is
retrieved from a hash table based on the header’s session identifier field. The ses-
sions are stored in the sessions hash table for later retrieval by the program. If the
session identifier is negative, the result of the modulus operator is negative, and
out-of-bounds elements of the sessions array are indexed and possibly written to,
which would probably be an exploitable condition.

As with the right-shift operator, unsigned and signed divide and modulus
operations can be distinguished easily in Intel assembly code. The mnemonic for
the unsigned division instruction is div and its signed counterpart is idiv. Table
6-10 shows the difference between signed and unsigned divide operations. Note that compilers often use right-shift operations rather than division when the divisor is a constant.

Table 6-10
Signed Versus Unsigned Divide Operations in Assembly

<table>
<thead>
<tr>
<th>Signed Divide Operations</th>
<th>Unsigned Divide Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, [ebp+8]</td>
<td>mov eax, [ebp+8]</td>
</tr>
<tr>
<td>mov ecx, [ebp+c]</td>
<td>mov ecx, [ebp+c]</td>
</tr>
<tr>
<td>cdq</td>
<td>cdq</td>
</tr>
<tr>
<td>idiv ecx</td>
<td>div ecx</td>
</tr>
<tr>
<td>ret</td>
<td>ret</td>
</tr>
</tbody>
</table>

Auditing Tip: Unexpected Results
Whenever you encounter a right shift, be sure to check whether the left operand is signed. If so, there might be a slight potential for a vulnerability. Similarly, look for modulus and division operations that operate with signed operands. If users can specify negative values, they might be able to elicit unexpected results.

Pointer Arithmetic

Pointers are usually the first major hurdle that beginning C programmers encounter, as they can prove quite difficult to understand. The rules involving pointer arithmetic, dereferencing and indirection, pass-by-value semantics, pointer operator precedence, and pseudo-equivalence with arrays can be challenging to learn. The following sections focus on a few aspects of pointer arithmetic that might catch developers by surprise and lead to possible security exposures.

Pointer Overview

You know that a pointer is essentially a location in memory—an address—so it’s a data type that’s necessarily implementation dependent. You could have strikingly different pointer representations on different architectures, and pointers could be implemented in different fashions even on the 32-bit Intel architecture. For example, you could have 16-bit code, or even a compiler that transparently supported custom virtual memory schemes involving segments. So assume this discussion uses the common architecture of GCC or vc++ compilers for userland code on Intel machines.

You know that pointers probably have to be unsigned integers because valid virtual memory addresses can range from 0x0 to 0xffffffff. That said, it seems slightly
odd when you subtract two pointers. Wouldn’t a pointer need to somehow repre-
sent negative values as well? It turns out that the result of the subtraction isn’t a
pointer at all; instead, it’s a signed integer type known as a **ptrdiff_t**.

Pointers can be freely converted into integers and into pointers of other types
with the use of casts. However, the compiler makes no guarantee that the resulting
pointer or integer is correctly aligned or points to a valid object. Therefore, pointers
are one of the more implementation-dependent portions of the C language.

**Pointer Arithmetic Overview**

When you do arithmetic with a pointer, what occurs? Here’s a simple example of
adding 1 to a pointer:

```c
short *j;
j=(short *)0x1234;
j = j + 1;
```

This code has a pointer to a short named *j*. It’s initialized to an arbitrary fixed
address, 0x1234. This is bad C code, but it serves to get the point across. As men-
tioned previously, you can treat pointers and integers interchangeably as long you
use casts, but the results depend on the implementation. You might assume that
after you add 1 to *j*, *j* is equal to 0x1235. However, as you probably know, this isn’t
what happens. *j* is actually 0x1236.

When C does arithmetic involving a pointer, it does the operation relative to
the size of the pointer’s target. So when you add 1 to a pointer to an object, the
result is a pointer to the next object of that size in memory. In this example, the
object is a short integer, which takes up 2 bytes (on the 32-bit Intel architecture),
so the short following 0x1234 in memory is at location 0x1236. If you subtract 1,
the result is the address of the short before the one at 0x1234, which is 0x1232. If
you add 5, you get the address 0x123e, which is the fifth short past the one at
0x1234.

Another way to think of it is that a pointer to an object is treated as an array
composed of one element of that object. So *j*, a pointer to a short, is treated like the
array short * j[1]*, which contains one short. Therefore, *j* + 2 would be equivalent
to &j[2]. Table 6-11 shows this concept.

<table>
<thead>
<tr>
<th>Pointer Expression</th>
<th>Array Expression</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>j - 2</td>
<td>&amp;j[-2]</td>
<td>0x1230</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1231</td>
</tr>
</tbody>
</table>
Now look at the details of the important pointer arithmetic operators, covered in the following sections.

**Addition**

The rules for pointer addition are slightly more restrictive than you might expect. You can add an integer type to a pointer type or a pointer type to an integer type, but you can’t add a pointer type to a pointer type. This makes sense when you consider what pointer addition actually does; the compiler wouldn’t know which pointer to use as the base type and which to use as an index. For example, look at the following operation:

```c
unsigned short *j;
unsigned long *k;

x = j + k;
```

This operation would be invalid because the compiler wouldn’t know how to convert j or k into an index for the pointer arithmetic. You could certainly cast j or k into an integer, but the result would be unexpected, and it’s unlikely someone would do this intentionally.

One interesting rule of C is that the subscript operator falls under the category of pointer addition. The C standard states that the subscript operator is equivalent to an expression involving addition in the following way:

$$E1[E2] \text{ is equivalent to } (*((E1)+(E2)))$$
With this in mind, look at the following example:

```c
char b[10];

b[4] = 'a';
```

The expression `b[4]` refers to the fifth object in the `b` character array. According to the rule, here’s the equivalent way of writing it:

```c
(*((b)+(4))) = 'a';
```

You know from your earlier analysis that `b + 4`, with `b` of type pointer to char, is the same as saying `&b[4]`; therefore, the expression would be like saying `(*(&b[4]))` or `b[4]`.

Finally, note that the resulting type of the addition between an integer and a pointer is the type of the pointer.

**Subtraction**

Subtraction has similar rules to addition, except subtracting one pointer from another is permissible. When you subtract a pointer from a pointer of the same type, you’re asking for the difference in the subscripts of the two elements. In this case, the resulting type isn’t a pointer but a `ptrdiff_t`, which is a signed integer type. The C standard indicates it should be defined in the `stddef.h` header file.

**Comparison**

Comparison between pointers works as you might expect. They consider the relative locations of the two pointers in the virtual address space. The resulting type is the same as with other comparisons: an integer type containing a 1 or 0.

**Conditional Operator**

The conditional operator (`?`) can have pointers as its last two operands, and it has to reconcile their types much as it does when used with arithmetic operands. It does this by applying all qualifiers either pointer type has to the resulting type.

**Vulnerabilities**

Few vulnerabilities involving pointer arithmetic have been widely publicized, at least in the sense being described here. Plenty of vulnerabilities that involve manipulation of character pointers essentially boil down to miscounting buffer sizes, and although they technically qualify as pointer arithmetic errors, they aren’t as subtle as pointer vulnerabilities can get. The more pernicious form of problems are those in which developers mistakenly perform arithmetic on pointers without realizing
that their integer operands are being scaled by the size of the pointer’s target. Consider the following code:

```c
int buf[1024];
int *b=buf;

while (havedata() && b < buf + sizeof(buf))
{
    *b++=parseInt(getdata());
}
```

The intent of `b < buf + sizeof(buf)` is to prevent `b` from advancing past `buf[1023]`. However, it actually prevents `b` from advancing past `buf[4092]`. Therefore, this code is potentially vulnerable to a fairly straightforward buffer overflow.

Listing 6-29 allocates a buffer and then copies the first path component from the argument string into the buffer. There’s a length check protecting the `wcscat` function from overflowing the allocated buffer, but it’s constructed incorrectly. Because the strings are wide characters, the pointer subtraction done to check the size of the input `(sep - string)` returns the difference of the two pointers in wide characters—that is, the difference between the two pointers in bytes divided by 2. Therefore, this length check succeeds as long as `(sep – string)` contains less than `(MAXCHARS * 2)` wide characters, which could be twice as much space as the allocated buffer can hold.

**Listing 6-29**

*Pointer Arithmetic Vulnerability Example*

```c
wchar_t *copy_data(wchar_t *string)
{
    wchar *sep, *new;
    int size = MAXCHARS * sizeof(wchar);

    new = (wchar *)xmalloc(size);
    *new = '\0';

    if(*string != '/'){
        wcscpy(new, '/');
        size -= sizeof(wchar_t);
    }

    sep = wstrchr(string, '/');

    if(!sep)
        sep = string + wcslen(string);
```
if (sep - string >= (size - sizeof(wchar_t))
{
    free(new);
    die("too much data");
}

*sep = '\0';

wcscat(new, string);

return new;

Auditing Tip

Pointer arithmetic bugs can be hard to spot. Whenever an arithmetic
operation is performed that involves pointers, look up the type of
those pointers and then check whether the operation agrees with
the implicit arithmetic taking place. In Listing 6-29, has sizeof() been
used incorrectly with a pointer to a type that’s not a byte? Has
a similar operation happened in which the developer assumed the
pointer type won’t affect how the operation is performed?

Other C Nuances

The following sections touch on features and dark corners of the C language where
security-relevant mistakes could be made. Not many real-world examples of these
vulnerabilities are available, yet you should still be aware of the potential risks.
Some examples might seem contrived, but try to imagine them as hidden beneath
layers of macros and interdependent functions, and they might seem more realistic.

Order of Evaluation

For most operators, C doesn’t guarantee the order of evaluation of operands or the
order of assignments from expression “side effects.” For example, consider this code:

printf("%d\n", i++, i++);

There’s no guarantee in which order the two increments are performed, and you’ll
find that the output varies based on the compiler and the architecture on which you
compile the program. The only operators for which order of evaluation is guaranteed
are &&, ||, ?, :, and ,. Note that the comma doesn’t refer to the arguments of a func-
tion; their evaluation order is implementation defined. So in something as simple as
the following code, there’s no guarantee that a() is called before b():

282
x = a() + b();

Ambiguous side effects are slightly different from ambiguous order of evaluation, but they have similar consequences. A side effect is an expression that causes the modification of a variable—an assignment or increment operator, such as ++. The order of evaluation of side effects isn’t defined within the same expression, so something like the following is implementation defined and, therefore, could cause problems:

a[i] = i++;

How could these problems have a security impact? In Listing 6-30, the developer uses the `getstr()` call to get the user string and pass string from some external source. However, if the system is recompiled and the order of evaluation for the `getstr()` function changes, the code could end up logging the password instead of the username. Admittedly, it would be a low-risk issue caught during testing.

Listing 6-30
Order of Evaluation Logic Vulnerability

```c
int check_password(char *user, char *pass)
{
    if (strcmp(getpass(user), pass))
    {
        logprintf("bad password for user %s\n", user);
        return -1;
    }
    return 0;
}
...
if (check_password(getstr(), getstr())
    exit(1);
```

Listing 6-31 has a `copy_packet()` function that reads a packet from the network. It uses the `GET32()` macro to pull an integer from the packet and advance the pointer. There’s a provision for optional padding in the protocol, and the presence of the padding size field is indicated by a flag in the packet header. So if `FLAG_PADDING` is set, the order of evaluation of the `GET32()` macros for calculating the `datasize` could possibly be reversed. If the padding option is in a fairly unused part of the protocol, an error of this nature could go undetected in production use.

Listing 6-31
Order of Evaluation Macro Vulnerability

```c
#define GET32(x) (*((unsigned int *)(x))++)

u_char *copy_packet(u_char *packet)
```
{  
  int *w = (int *)packet;
  unsigned int hdrvar, datasize;

  /* packet format is hdr var, data size, padding size */
  hdrvar = GET32(w);

  if (hdrvar & FLAG_PADDING)  
    datasize = GET32(w) - GET32(w);
  else  
    datasize = GET32(w);

  ...
}

Structure Padding

One somewhat obscure feature of C structures is that structure members don’t have
to be laid out contiguously in memory. The order of members is guaranteed to fol-
low the order programmers specify, but structure padding can be used between
members to facilitate alignment and performance needs. Here’s an example of a
simple structure:

struct bob
{
  int a;
  unsigned short b;
  unsigned char c;
};

What do you think sizeof(bob) is? A reasonable guess is 7; that’s sizeof(a) +
sizeof(b) + sizeof(c), which is 4 + 2 + 1. However, most compilers return 8
because they insert structure padding! This behavior is somewhat obscure now, but
it will definitely become a well-known phenomenon as more 64-bit code is intro-
duced because it has the potential to affect this code more acutely. How could it
have a security consequence? Consider Listing 6-32.

Listing 6-32
Structure Padding in a Network Protocol

struct netdata
{
  unsigned int query_id;
  unsigned short header_flags;
}
unsigned int sequence_number;
};

int packet_check_replay(unsigned char *buf, size_t len)
{
    struct netdata *n = (struct netdata *)buf;

    if (((ntohl(n->sequence_number) <= g_last_sequence_number)
        return PARSE_REPLAYATTACK;

    // packet is safe - process
    return PARSE_SAFE;
}

On a 32-bit big-endian system, the netdata structure is likely to be laid out as shown in Figure 6-9. You have an unsigned int, an unsigned short, 2 bytes of padding, and an unsigned int for a total structure size of 12 bytes. Figure 6-10 shows the traffic going over the network, in network byte order. If developers don’t anticipate the padding being inserted in the structure, they could be misinterpreting the network protocol. This error could cause the server to accept a replay attack.

The possibility of making this kind of mistake increases with 64-bit architectures. If a structure contains a pointer or long value, the layout of the structure in memory will most likely change. Any 64-bit value, such as a pointer or long int, will take up twice as much space as on a 32 bit-system and have to be placed on a 64-bit alignment boundary.
The contents of the padding bits depend on whatever happens to be in memory when the structure is allocated. These bits could be different, which could lead to logic errors involving memory comparisons, as shown in Listing 6-33.

**Listing 6-33**  
*Example of Structure Padding Double Free*

```c
text sh
{
    void *base;
    unsigned char code;
    void *descptr;
};

void free_sechdrs(struct sh *a, struct sh *b)
{
    if (!memcmp(a, b, sizeof(a)))
    {
        /* they are equivalent */
        free(a->descptr);
        free(a->base);
        free(a);
        return;
    }

    free(a->descptr);
    free(a->base);
    free(a);
    free(b->descptr);
    free(b->base);
    free(b);
    return;
}
```

If the structure padding is different in the two structures, it could cause a double-free error to occur. Take a look at Listing 6-34.

**Listing 6-34**  
*Example of Bad Counting with Structure Padding*

```c
text hdr
{
    int flags;
    short len;
};

text hdropt
{
    char opt1;
```
char optlen;
char descl;
};

struct msghdr
{
    struct hdr h;
    struct hdropt o;
};

struct msghdr *form_hdr(struct hdr *h, struct hdropt *o)
{
    struct msghdr *m=xmalloc(sizeof *h + sizeof *o);
    memset(m, 0, sizeof(struct msghdr));

    ... 

The size of hdropt would likely be 3 because there are no padding requirements for alignment. The size of hdr would likely be 8 and the size of msghdr would likely be 12 to align the two structures. Therefore, memset would write 1 byte past the allocated data with a \0.

**Precedence**

When you review code written by experienced developers, you often see complex expressions that seem to be precariously void of parentheses. An interesting vulnerability would be a situation in which a precedence mistake is made but occurs in such a way that it doesn’t totally disrupt the program.

The first potential problem is the precedence of the bitwise & and | operators, especially when you mix them with comparison and equality operators, as shown in this example:

if ( len & 0x80000000 != 0) 
    die("bad len!");

if (len < 1024) 
    memcpy(dst, src, len);

The programmers are trying to see whether len is negative by checking the highest bit. Their intent is something like this:

if ( (len & 0x800000000) != 0) 
    die("bad len!");
What’s actually rendered into assembly code, however, is this:

```c
if ( len & (0x80000000 != 0))
    die("bad len!");
```

This code would evaluate to `len & 1`. If `len`’s least significant bit isn’t set, that test would pass, and users could specify a negative argument to `memcpy()`.

There are also potential precedence problems involving assignment, but they aren’t likely to surface in production code because of compiler warnings. For example, look at the following code:

```c
if (len = getlen() > 30)
    snprintf(dst, len - 30, "%s", src)
```

The authors intended the following:

```c
if ((len = getlen()) > 30)
    snprintf(dst, len - 30, "%s", src)
```

However, they got the following:

```c
if (len = (getlen() > 30))
    snprintf(dst, len - 30, "%s", src)
```

`len` is going to be 1 or 0 coming out of the `if` statement. If it’s 1, the second argument to `snprintf()` is -29, which is essentially an unlimited string.

Here’s one more potential precedence error:

```c
int a = b + c >> 3;
```

The authors intended the following:

```c
int a = b + (c >> 3);
```

As you can imagine, they got the following:

```c
int a = (b + c) >> 3;
```

### Macros/Preprocessor

C’s preprocessor could also be a source of security problems. Most people are familiar with the problems in a macro like this:

```c
#define SQUARE(x) x*x
```

Chapter 6—C Language Issues

288
If you use it as follows:

\[ y = \text{SQUARE}(z + t); \]

It would evaluate to the following:

\[ y = z + t \times z + t; \]

That result is obviously wrong. The recommended fix is to put parentheses around the macro and the arguments so that you have the following:

\[
\text{#define SQUARE(x) } ((x)*(x))
\]

You can still get into trouble with macros constructed in this way when you consider order of evaluation and side-effect problems. For example, if you use the following:

\[ y = \text{SQUARE}(j++); \]

It would evaluate to

\[ y = ((j++)*(j++)); \]

That result is implementation defined. Similarly, if you use the following:

\[ y = \text{SQUARE} \text{ (getint());} \]

It would evaluate to

\[ y = ((\text{getint()})*(\text{getint()})); \]

This result is probably not what the author intended. Macros could certainly introduce security issues if they’re used in way outside mainstream use, so pay attention when you’re auditing code that makes heavy use of them. When in doubt, expand them by hand or look at the output of the preprocessor pass.

**Typos**

Programmers can make many simple typographic errors that might not affect program compilation or disrupt a program’s runtime processes, but these typos could lead to security-relevant problems. These errors are somewhat rare in production code, but occasionally they crop up. It can be entertaining to try to spot typos in code. Possible typographic mistakes have been presented as a series of challenges. Try to spot the mistake before reading the analysis.
Challenge 1

while (*src && left)
{
    *dst++=*src++;
    if (left = 0)
        die("badlen");
    left--;
}

The statement if (left = 0) should read if (left == 0).

In the correct version of the code, if left is 0, the loop detects a buffer overflow attempt and aborts. In the incorrect version, the if statement assigns 0 to left, and the result of that assignment is the value 0. The statement if (0) isn't true, so the next thing that occurs is the left--; statement. Because left is 0, left-- becomes a negative 1 or a large positive number, depending on left's type. Either way, left isn't 0, so the while loop continues, and the check doesn't prevent a buffer overflow.

Challenge 2

int f;

f=get_security_flags(username);
if (f = FLAG_AUTHENTICATED)
{
    return LOGIN_OK;
}
return LOGIN_FAILED;

The statement if (f = FLAG_AUTHENTICATED) should read as follows:
if (f == FLAG_AUTHENTICATED)

In the correct version of the code, if users' security flags indicate they're authenticated, the function returns LOGIN_OK. Otherwise, it returns LOGIN_FAILED.

In the incorrect version, the if statement assigns whatever FLAG_AUTHENTICATED happens to be to f. The if statement always succeeds because FLAG_AUTHENTICATED is some nonzero value. Therefore, the function returns LOGIN_OK for every user.
**Challenge 3**

```c
for (i==5; src[i] && i<10; i++)
{
    dst[i-5]=src[i];
}
```

The statement `for (i==5; src[i] && i<10; i++)` should read as follows:

```c
for (i=5; src[i] && i<10; i++)
```

In the correct version of the code, the `for` loop copies 4 bytes, starting reading from `src[5]` and starting writing to `dst[0]`. In the incorrect version, the expression `i==5` evaluates to true or false but doesn’t affect the contents of `i`. Therefore, if `i` is some value less than 10, it could cause the `for` loop to write and read outside the bounds of the `dst` and `src` buffers.

**Challenge 4**

```c
if (get_string(src) &&
    check_for_overflow(src) && copy_string(dst,src))
    printf("string safely copied\n");
```

The `if` statement should read like so:

```c
if (get_string(src) &&
    check_for_overflow(src) && copy_string(dst,src))
```

In the correct version of the code, the program gets a string into the `src` buffer and checks the `src` buffer for an overflow. If there isn’t an overflow, it copies the string to the `dst` buffer and prints “string safely copied.”

In the incorrect version, the `&` operator doesn’t have the same characteristics as the `&&` operator. Even if there isn’t an issue caused by the difference between logical and bitwise `AND` operations in this situation, there’s still the critical problem of short-circuit evaluation and guaranteed order of execution. Because it’s a bitwise `AND` operation, both operand expressions are evaluated, and the order in which they are evaluated isn’t necessarily known. Therefore, `copy_string()` is called even if `check_for_overflow()` fails, and it might be called before `check_for_overflow()` is called.

**Challenge 5**

```c
if (len > 0 && len <= sizeof(dst));
    memcpy(dst, src, len);
```
The if statement should read like so:

if (len > 0 && len <= sizeof(dst))

In the correct version of the code, the program performs a `memcpy()` only if the length is within a certain set of bounds, therefore preventing a buffer overflow attack. In the incorrect version, the extra semicolon at the end of the if statement denotes an empty statement, which means `memcpy()` always runs, regardless of the result of length checks.

**Challenge 6**

```c
char buf[040];

snprintf(buf, 40, "%s", userinput);
```

The statement `char buf[040];` should read `char buf[40];`.

In the correct version of the code, the program sets aside 40 bytes for the buffer it uses to copy the user input into. In the incorrect version, the program sets aside 32 bytes. When an integer constant is preceded by 0 in C, it instructs the compiler that the constant is in octal. Therefore, the buffer length is interpreted as 040 octal, or 32 decimal, and `snprintf()` could write past the end of the stack buffer.

**Challenge 7**

```c
if (len < 0 || len > sizeof(dst)) /* check the length */
    die("bad length!");

/* length ok */

memcpy(dst, src, len);
```

The if statement should read like so:

if (len < 0 || len > sizeof(dst)) /* check the length */

In the correct version of the code, the program checks the length before it carries out `memcpy()` and calls `abort()` if the length is out of the appropriate range.

In the incorrect version, the lack of an end to the comment means `memcpy()` becomes the target statement for the if statement. So `memcpy()` occurs only if the length checks fail.
Challenge 8

```c
if (len > 0 && len <= sizeof(dst))
    copiedflag = 1;
    memcpy(dst, src, len);

if (!copiedflag)
    die("didn't copy");
```

The first if statement should read like so:

```c
if (len > 0 && len <= sizeof(dst))
{
    copiedflag = 1;
    memcpy(dst, src, len);
}
```

In the correct version, the program checks the length before it carries out `memcpy()`. If the length is out of the appropriate range, the program sets a flag that causes an abort.

In the incorrect version, the lack of a compound statement following the `if` statement means `memcpy()` is always performed. The indentation is intended to trick the reader’s eyes.

Challenge 9

```c
if (!strncmp(src, "magicword", 9))
    // report_magic(1);
```

```c
if (len < 0 || len > sizeof(dst))
    assert("bad length!");

/* length ok */
```

```c
memcpy(dst, src, len);
```

The `report_magic(1)` statement should read like so:

```c
    // report_magic(1);
```
In the correct version, the program checks the length before it performs `memcpy()`. If the length is out of the appropriate range, the program sets a flag that causes an abort.

In the incorrect version, the lack of a compound statement following the `magicword if` statement means the length check is performed only if the `magicword` comparison is true. Therefore, `memcpy()` is likely always performed.

**Challenge 10**

```c
l = msg_hdr.msg_len;
frag_off = msg_hdr.frag_off;
frag_len = msg_hdr.frag_len;

...

if (frag_len > (unsigned long)max)
{
    al=SSL_AD_ILLEGAL_PARAMETER;
    SSLerr(SSL_F_DTLS1_GET_MESSAGE_FRAGMENT,
            SSL_R_EXCESSIVE_MESSAGE_SIZE);
    goto f_err;
}

if (frag_len + s->init_num >
    (INT_MAX - DTLS1_HM_HEADER_LENGTH))
{
    al=SSL_AD_ILLEGAL_PARAMETER;
    SSLerr(SSL_F_DTLS1_GET_MESSAGE_FRAGMENT,
            SSL_R_EXCESSIVE_MESSAGE_SIZE);
    goto f_err;
}

if (frag_len &
    !BUF_MEM_grow_clean(s->init_buf, (int)frag_len +
                        DTLS1_HM_HEADER_LENGTH + s->init_num))
{
    SSLerr(SSL_F_DTLS1_GET_MESSAGE_FRAGMENT,
            ERR_R_BUF_LIB);
    goto err;
}
```
if ( s->d1->r_msg_hdr.frag_off == 0)
{
    s->s3->tmp.message_type = msg_hdr.type;
    s->d1->r_msg_hdr.type = msg_hdr.type;
    s->d1->r_msg_hdr.msg_len = l;
    /* s->d1->r_msg_hdr.seq = seq_num; */
}
/* XDTLS: resurrect this when restart is in place */
s->state=stn;
/* next state (stn) */
p = (unsigned char *)s->init_buf->data;

if ( frag_len > 0)
{
    i=s->method->ssl_read_bytes(s,SSL3_RT_HANDSHAKE,
        &p[s->init_num],
        frag_len,0);
    /* XDTLS: fix this—message fragments cannot
        span multiple packets */
    if (i <= 0)
    {
        s->rwstate=SSL_READING;
        *ok = 0;
        return i;
    }
}
else
    i = 0;

Did you spot the bug? There is a mistake in one of the length checks where the developers use a bitwise AND operator (&) instead of a logical AND operator (&&). Specifically, the statement should read:
if ( frag_len &&
    !BUF_MEM_grow_clean(s->init_buf, (int)frag_len +
                        DTLS1_HM_HEADER_LENGTH + s->init_num))

    This simple mistake could lead to memory corruption if the BUF_MEM_grow_clean() function were to fail. This function returns 0 upon failure, which will be set to 1 by the logical not operator. Then, a bitwise AND operation with frag_len will occur. So, in the case of failure, the malformed statement is really doing the following:

    if(frag_len & 1)
    {
        SSLerr(...);
    }

Summary

This chapter has covered nuances of the C programming language that can lead to subtle and complex vulnerabilities. This background should enable you to identify problems that can occur with operator handling, type conversions, arithmetic operations, and common C typos. However, the complex nature of this topic does not lend itself to complete understanding in just one pass. Therefore, refer back to this material as needed when conducting application assessments. After all, even the best code auditor can easily miss subtle errors that could result in severe vulnerabilities.
Index

Symbols
/bin directory (UNIX), 463
/etc directory (UNIX), 463
/home directory (UNIX), 463
%m format specifier, 423
/sbin directory (UNIX), 463
/var directory (UNIX), 463

A
AASP (Active Server Pages), 1013
Abstract Syntax Notation (ASN.1). See ASN.1 (Abstract Syntax Notation)
Abstraction, software design, 27
ACC (allocation-check-copy) logs, 363
auditing, 362-369
data assumptions, 365-366
order of action, 366-367
unanticipated conditions, 364-365
Accept header field (HTTP), 1018
Accept-Charset header field (HTTP), 1018
Accept-Encoding header field (HTTP), 1018
Accept-Language header field (HTTP), 1018
Accept-Ranges header field (HTTP), 1018
access control, 1057-1058
ASP.NET, 1122
COM (Component Object Model), 734-736
vulnerabilities, 69-70
access control entries (ACEs). See ACEs (access control entries)
access control policy, 38
access masks, Windows NT, security descriptors, 648-649
access tokens, Windows NT sessions, 639-640
contexts, 644-645
group lists, 641
impersonation, 647
privileges, 640-641
restricted tokens, 642-644
SAFER (Software Restriction Policies) API, 644
access() function, 527
accountability, common vulnerabilities, 40-41
accuracy, software design, 32
ACEs (access control entries), 647
flags, 650
orders, 653
ACFs (application configuration files), RPCs (Remote Procedure Calls), 710
ACLs (access control lists), 647
low-level ACL control, 650
permissions, auditing, 652-653
Windows NT, inheritance, 649
activation, DCOM objects, 734
activation records, runtime stack, 170
active FTP, 900
Active Server Pages (ASP). See ASP (Active Server Pages)
Active X controls, 749, 753-754
COM (Component Object Model), security, 749-754
kill bit, 752
signing, 750
site-restricted controls, 752
threading, 753
ActiveX Data Objects (ADO), 1113-1115
address space layout randomization (ASLR). See ASLR (address space layout randomization)
<table>
<thead>
<tr>
<th>Addresses</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP addresses, 832-834</td>
<td>1029</td>
</tr>
<tr>
<td>subnet addresses, 834</td>
<td>1030</td>
</tr>
<tr>
<td>AdjustTokenGroups() function, 643</td>
<td></td>
</tr>
<tr>
<td>AdjustTokenPrivileges() function, 643</td>
<td></td>
</tr>
<tr>
<td>ADO (ActiveX Data Objects), 1113-1115</td>
<td></td>
</tr>
<tr>
<td>ADT (abstract data type), stacks, 169</td>
<td></td>
</tr>
<tr>
<td>Age header field (HTTP), 1018</td>
<td></td>
</tr>
<tr>
<td>Aitel, Dave, 158</td>
<td></td>
</tr>
<tr>
<td>AIX, 460</td>
<td></td>
</tr>
<tr>
<td>AJAX (Asynchronous JavaScript and XML), 1085</td>
<td></td>
</tr>
<tr>
<td>Algorithms</td>
<td></td>
</tr>
<tr>
<td>analyzing, CC (code comprehension), 116</td>
<td></td>
</tr>
<tr>
<td>encryption, 41-42</td>
<td></td>
</tr>
<tr>
<td>block ciphers, 42</td>
<td></td>
</tr>
<tr>
<td>common vulnerabilities, 43-45</td>
<td></td>
</tr>
<tr>
<td>exchange algorithms, 43</td>
<td></td>
</tr>
<tr>
<td>IV (initialization vector), 42</td>
<td></td>
</tr>
<tr>
<td>stream ciphers, 42</td>
<td></td>
</tr>
<tr>
<td>hashing algorithms, 326</td>
<td></td>
</tr>
<tr>
<td>software design, 26-27</td>
<td></td>
</tr>
<tr>
<td>alloc() function, 318</td>
<td></td>
</tr>
<tr>
<td>allocating 0 bytes, 370-371</td>
<td></td>
</tr>
<tr>
<td>allocation functions, auditing, 369-377</td>
<td></td>
</tr>
<tr>
<td>allocation-check-copy (ACC) logs. See ACC</td>
<td></td>
</tr>
<tr>
<td>(allocation-check-copy) logs</td>
<td></td>
</tr>
<tr>
<td>allocator scorecards, 377-379</td>
<td></td>
</tr>
<tr>
<td>Allocator with Header Data Structure listing (7-39), 372</td>
<td></td>
</tr>
<tr>
<td>Allocator-Rounding Vulnerability listing (7-38), 372</td>
<td></td>
</tr>
<tr>
<td>Allow header field (HTTP), 1018</td>
<td></td>
</tr>
<tr>
<td>Allowed header field (HTTP), 1018</td>
<td></td>
</tr>
<tr>
<td>analysis phase, code review, 93, 106-108</td>
<td></td>
</tr>
<tr>
<td>findings summary, 106</td>
<td></td>
</tr>
<tr>
<td>analyzing</td>
<td></td>
</tr>
<tr>
<td>algorithms, CC (code comprehension), 116</td>
<td></td>
</tr>
<tr>
<td>classes, CC (code comprehension), 116-117</td>
<td></td>
</tr>
<tr>
<td>modules, CC (code comprehension), 114-116</td>
<td></td>
</tr>
<tr>
<td>objects, CC (code comprehension), 116-117</td>
<td></td>
</tr>
<tr>
<td>Anderson, J.S., 459</td>
<td></td>
</tr>
<tr>
<td>anonymous pipes, Windows NT, 698</td>
<td></td>
</tr>
<tr>
<td>antivirus applications, 82-83</td>
<td></td>
</tr>
<tr>
<td>antisniff tool, vulnerabilities, 249-255</td>
<td></td>
</tr>
<tr>
<td>Antisniff v1.0 Vulnerability listing (6-8), 250</td>
<td></td>
</tr>
<tr>
<td>Antisniff v1.1 Vulnerability listing (6-9), 251</td>
<td></td>
</tr>
<tr>
<td>Antisniff v1.1.1 Vulnerability listing (6-10), 252</td>
<td></td>
</tr>
<tr>
<td>Antisniff v1.1.2 Vulnerability listing (6-11), 253</td>
<td></td>
</tr>
<tr>
<td>Apache, Struts framework, 1008</td>
<td></td>
</tr>
<tr>
<td>Apache 1.3.29/2.X mod_rewrite Off-by-one</td>
<td></td>
</tr>
<tr>
<td>Vulnerability listing (7-19), 332</td>
<td></td>
</tr>
<tr>
<td>Apache API, 1011</td>
<td></td>
</tr>
<tr>
<td>Apache mod_day CDATA Parsing Vulnerability listing (7-1), 298</td>
<td></td>
</tr>
</tbody>
</table>

Apache mod_php Nonterminating Buffer Vulnerability listing (7-18), 331
APCs (asynchronous procedure calls), 765
APIs (application programming interfaces)
  Apache API, 1011
  ISAPI (Internet Server Application Programming Interface), 1010
  NSAPI (Netscape Server Application Programming Interface), 1010
Appel, Andrew W., 80
ApplID keys, 728
application access, categories, 95-96
application architecture modeling, 53-66
application identity, DCOM (Distributed Component Object Model), 732-733
application IDs, COM (Component Object Model), 728
application layer, network segmentation, 87-88
application manifests, 657
application protocols, 921
  ASN.1 (Abstract Syntax Notation), 972-974
  BER (Basic Encoding Rules), 975-979
  CER (Canonical Encoding Rules), 976-979
  DER (Distinguished Encoding Rules), 977, 979
  PER (Packed Encoding Rules), 979-983
  XER (XML Encoding Rules), 983-984
auditing, 922-937
  data type matching, 927-934
  data verification, 935
  documentation collection, 922-923
  identifying elements, 923-927
  system resource access, 935-937
DNS (Domain Name System), 984, 989-990
  headers, 991-992
  length variables, 996-1002
  name servers, 986-987
  names, 993-996
  packets, 991
  question structure, 992
  request traffic, 989
  resolvers, 986-987
  resource records, 984-985, 988, 993
  spoofing, 1002-1005
  zones, 986-987
HTTP (Hypertext Transfer Protocol), 937-948
  header parsing, 937-938
  posting data, 942-948
  resource access, 940-941
  utility functions, 941-942
ISAKMP (Internet Security Association and Key Management Protocol), 948
  encryption vulnerabilities, 971-972
  headers, 949-952
  payloads, 952-971
application review, 91-93
application review phase, 93, 97-98, 103-105
bottom-up approach, 100
hybrid approach, 100-101
iterative process, 98-99
peer reviews, 106
planning, 101-103
reevaluation, 105
status checks, 105
top-down approach, 99
working papers, 103-104
code auditing, 111, 133, 147
binary navigation tools, 155-157
CC (code comprehension) strategies, 112-119
CP (candidate point) strategies, 112, 119-128
debuggers, 151-154
dependency analysis, 135-136
desk checking, 137-139
dG (design generalization) strategies, 112, 128-133
fuzz testing tools, 157-158
internal flow analysis, 133-135
OpenSSH case study, 158-164
rereading code, 136-137
scorecard, 112
source code navigators, 148-151
subsystem analysis, 135-136
test cases, 139-147
code navigation, 109
external flow sensitivity, 109-110
tracing, 111
documentation and analysis phase, 93, 106-108
findings summary, 106
preassessment phase, 93
application access, 95-96
information collection, 96
scoping, 94
process outline, 93
remediation support phase, 93, 108-109
application-specific CPs (candidate points), 128
applications
attack surfaces, 68
COM (Component Object Model) applications, registration, 741-743
DCOM (Distributed Component Object Model) applications, auditing, 741-749
reverse-engineering applications, 924-927
RPC (Remote Procedure Call) applications, auditing, 722-724
Web applications. See Web applications
Applied Cryptography, 41
appSettings section, ASP.NET, 1123
apr_palloc() function, 299
arbitrary file accesses, junction points, 678-680
argument promotions, 232
arguments, functions, auditing, 360-362
arithmetic
C programming language
arithmetic boundary conditions, 211-223
signed integer boundaries, 220-223
unsigned integer boundaries, 213-220
modular arithmetic, 214
pointers, 278-280
arithmetic boundaries, variables, auditing, 316-319
arithmetic boundary conditions, C programming language, 211-223
numeric overflow conditions, 211-212
numeric underflow conditions, 212
numeric wrapping, 212
signed integers, 220-223
unsigned integers, 213-220
arithmetic shift, 273
Arithmetic Vulnerability Example in the Parent Function listing (7-10), 318
Arithmetic Vulnerability Example listing (7-9), 317
ASLR (address space layout randomization), 194
operational vulnerabilities, preventing, 78
ASN.1 (Abstract Syntax Notation), 972-974
BER (Basic Encoding Rules), 975-979
CER (Canonical Encoding Rules), 976-979
DER (Distinguished Encoding Rules), 977, 979
PER (Packed Encoding Rules), 979-983
XER (XML Encoding Rules), 983-984
ASP (Active Server Pages), 1113
configuration settings, 1118
cross-site scripting, 1118
file access, 1115
file inclusion, 1116-1117
inline evaluation, 1117-1118
shell invocation, 1115
SQL injection queries, 1113-1115
ASP.NET, 1118
configuration settings, 1121-1123
cross-site scripting, 1121
file access, 1119-1120
file inclusion, 1120
inline evaluation, 1121
shell invocation, 1120
SQL injection queries, 1118-1119
assessments
applications, 91
code, 92-93
application review phase, 93, 97-106
code auditing, 111-133
code navigation, 109-111
documentation and analysis phase, 93, 106-108
preassessment phase, 93-96
process outline, 93
remediation support phase, 93, 108-109
assets, information collection, 50
assignment operators, C programming language, type conversions, 231-232
asymmetric encryption, 42
Asynchronous JavaScript and XML (AJAX), 1085
asynchronous procedure calls (APCs). See APCs (asynchronous procedure calls)
asynchronous-safe code, reentrancy, 757-759
asynchronous-safe function, signals, 791-797, 800-801, 804-809
ATL (Active Template Library), DCOM (Distributed Component Object Model), 740
atomicity, 756
attack surfaces
applications, 68
firewalls, 895
attack trees, 59-62
attack vectors, high-level attack vectors, OpenSSH, 162-164
attacks
attack surfaces, applications, 68
attack trees, 59-62
bait-and-switch attacks, 47
blind data injection attacks, 880
blind reset attacks, 879-880
cryogenic sleep attacks, 545-546
DoS (denial of service) attacks, 48
name validation, 931-932
environmental attacks, 21-22
exceptional conditions, 22
homographic attacks, 450
node types, 60
second-order injection attacks, 409
shatter attacks, 694-697
SHE (structured exception handling) attacks, 178-180
SMB relay attacks, 688
spoofing attacks, 72
DNS (Domain Name System), 1002-1005
firewalls, 914-920
terminal attacks, 609-610
attributes
objects, uninitialized attributes, 314-315
UNIX processes, 572-611
file descriptors, 580-591
resource limits, 574-580
retention, 573-574
audit logs, function audit logs, 339-340
auditing, 10
application protocols, 922-937
data type matching, 927-934
data verification, 935
documentation collection, 922-923
identifying elements, 923-927
system resource access, 935-937
black box testing, compared, 11-13
code, 111, 133, 147
binary navigation tools, 155-157
CC (code comprehension) strategies, 112-119
CP (candidate point) strategies, 112, 119-128
debuggers, 151-154
dependency analysis, 135-136
desk checking, 137-139
DG (design generalization) strategies, 112, 128-133
fuzz testing tools, 157-158
internal flow analysis, 133-135
OpenSSH case study, 158-164
rereading code, 136-137
scorecard, 112
SDLC (Systems Development Life Cycle), 13
source code navigators, 148-151
subsystem analysis, 135-136
test cases, 139-147
code-editing situations, 9
COM (Component Object Model) applications, interfaces, 743-749
control flow, 326-339
flow transfer statements, 336
looping constructs, 327-336
switch statements, 337-339
DCOM (Distributed Component Object Model) applications, 741-749
file opens, Windows NT, 674-675
functions, 339
argument meaning, 360-362
audit logs, 339-340
return value testing, 340-350
side-effects, 351, 353-359
hidden fields, 1036
importance of, 9, 11
memory management, 362
ACC (allocation-check-copy) logs, 362-369
allocation functions, 369-377
allocator scorecards, 377-379
double-frees, 379-385
error domains, 378-379
permissions, ACLs, 652-653
RPC applications, 722-724
running code, 567
UNIX privileges, management code, 488-490
variables, 298-326
arithmetic boundaries, 316-319
initialization, 312-315
lists, 321-326
object management, 307-312
relationships, 298-307
structure management, 307-312
tables, 321-326
type confusion, 319-321
Web applications, 1078-1081
  activities to isolate, 1079
  avoiding assumptions, 1080
  black box testing, 1079
  enumerating functionality, 1081
  goals, 1081
  multiple approaches, 1080
  reverse-engineering, 1081
  testing and experimentation, 1080-1081
authenticate() function, 177
authentication, 36
  common vulnerabilities, 36
  insufficient validation, 38
  untrustworthy credentials, 37
HTTP authentication, 1033-1036, 1056-1057
RPC servers, 714-716
RPCs (Remote Procedure Calls), UNIX, 623-624
Web-based applications, 75
authentication files, OpenSSH, 161
authorization, 38, 1057-1058
ASP.NET, 1122
  common vulnerabilities, 39
Authorization header field (HTTP), 1018
AUTH_TYPE (environment variable), 1088
automated source analysis tools, code audits, CP candidate point) strategy, 120-122
automatic threat modeling, 65
automation objects, COM (Component Object Model), 729
  fuzz testing, 749
automation servers, 729
availability, 48
  common vulnerabilities, 48-49
  expectations of, 9
back-tracing code, 111
bait-and-switch attacks, 47
Bansal, Altin, 235
Bellovin, Steve, 891
BER (Basic Encoding Rules), ASN.1 (Abstract Syntax Notation), 975-979
Bercegay, James, 1101
big-endian architecture, bytes, ordering, 209
/bin directory (UNIX), 463
binary audits, COM (Component Object Model), 743-749
binary bitwise operators, 243
binary encoding, C programming language, 207-208
binary layout (Windows), imports, 70
binary navigation tools, code auditing, 155-157
binary notation
  positive decimal integers, converting to, 207
  positive numbers, converting to decimal, 207
  binary protocols, data types, matching, 927-932
binary-only application access, 95
Bind 9.2.1 Resolver Code gethostans() Vulnerability listing (7-2), 300
binding endpoints, RPC servers, 712-714
bindings, 706
BinNavi binary navigation tool, 157
Bishop, Matt, 5
bit fields, C programming language, 205
bitmasks, permissions, 495-497
bitwise shift operators, C programming language, 236-237
black box analysis, 118
black box generated CPs (candidate points), 123-128
black box hits, tracing, 117-119
black box testing, 1079
  auditing, compared, 11-13
black-list filters, metacharacters, 435-436
blind connection spoofing, TCP streams, 876-879
blind data injection attacks, TCP streams, 880
blind reset attacks, TCP streams, 879-880
block ciphers, 42
boot files, UNIX, 511
bottom-up approach, application review, 100
bottom-up decomposition, 27
Bouchereine, Pascal, 877
boundaries, trust boundaries, 28
  complex trust boundaries, 30
  simple trust boundaries, 28-30
boundary conditions, sequence numbers, TCP (Transmission Control Protocol), 888
boundary descriptor objects, Windows NT, 631
bounded string functions, 393-400
Break Statement Omission Vulnerability listing (7-23), 337
break statements, omissions, 337-338
Bret-Mounet, Frederic, 749
Brown, Keith, 637
BSD linux, 459
  securelevels, 492
  setenv() function, 576-577
BUF-MEM_grow() function, 311-312
buffer overflow, text-based protocols, 933-934
Buffer Overflow in NSS Library’s ssl2_HandleClientHelloMessage listing (7-34), 365
buffer overflows, 168-169
  global overflows, 186
  heap overflows, 183-186
  off-by-one errors, 180-183
  process memory layout, 169
  SHE (structured exception handling) attacks, 178-180
  stack overflows, 169-178
  static overflows, 186
buffer subsystem, SSH1 server, code audits, 160
buffers, OpenSSH, vulnerabilities, 307-310
bugs, software, 4-5
INDEX

business logic, 26-27, 1041
business tier (Web applications), 1042-1044
byte order, C programming language, 209
bytes, overwriting, 198-199

c
C programming language, 204
arithmetic boundary conditions, 211-223
binary encoding, 207-208
bit fields, 205
bitwise shift operators, 236-237
byte order, 209
character types, 205
data storage, 204-211
floating types, 205
format strings, 422-425
function invocations, 237-238
implementation defined behavior, 204
integer types, 205-206
macros, 288-289
numeric wrapping, 212
objects, 205
operands, order of evaluation, 282-283
operators, 233, 271-277
right shift, 272-277
size, 271-272
pointers, 277-282
arithmetic, 278-280
vulnerabilities, 280-282
precedence, 287-288
preprocessor, 288-289
security, 1075
signed integers, boundaries, 220-223
standards, 204
stdio file interface, 547-557
string handling, 388-407
structure padding, 284-287
switch statements, 237
type conversions, 223-248
assignment operators, 231-232
comparisons, 265-270
conversion rules, 225-231
default type conversions, 224
explicit type conversions, 224
floating point types, 230-231
function prototypes, 232
implicit type conversions, 224
integer promotions, 233-238
narrowing, 227-228
sign extensions, 248-265
simple conversions, 231-232
typecasts, 231
usual arithmetic conversions, 238-245
value preservation, 225-226
vulnerabilities, 246-270
widening, 226-227
types, 204-207
typos, 289-296
unary operator, 236
unary + operator, 235
unary - operator, 235
undefined behavior, 204
unsigned integers, boundaries, 213-220
C Programming Language, The, 204
C Rationale document, 204
C++ programming language, EH (exception handling), 179
Cache-Control header field (HTTP), 1018
calling conventions, functions, 173
canonicalization, files, Windows NT, 663-666
capabilities, Linux, 492-494
carry flags (CFs), 214
CAS (code access security), 6
case sensitivity, Windows NT filenames, 673
CBC (cipher block chaining) mode cipher, 42
CC (code comprehension) strategies, code audits, 112-119
algorithm analysis, 116
black box hit traces, 117-119
class analysis, 116-117
module analysis, 114-116
object analysis, 116-117
trace malicious input, 113-114
CER (Canonical Encoding Rules), ASN.1 (Abstract Syntax Notation), 976-979
Certificate Payload Integer Underflow in CheckPoint ISAKMP listing (16-2), 954
certificate payloads, ISAKMP (Internet Security Association and Key Management Protocol), 963-964
certificate request payloads, ISAKMP (Internet Security Association and Key Management Protocol), 964
CFML (ColdFusion Markup Language), 1013
CFs (carry flags), 214
cgi (Common Gateway Interface), 1009-1010, 1086
environment variables, 1087-1093
indexed queries, 1086-1087
chain of trust relationships, 30-31
Challenge-Response Integer Overflow Example in OpenSSH 3.1 listing (6-3), 216
change monitoring, 83
Character Black-List Filter listing (8-22), 435
character equivalence, Unicode, 456-457
character expansion, text strings, 401
Character Expansion Buffer Overflow listing (8-4), 401
character sets, 446
character stripping vulnerabilities, metacharacters, filtering, 437-439
case types, C programming language, 205
INDEX

Character White-List Filter listing (8-23), 436
Charge-To header field (HTTP), 1018
checked build application access, 95
checkForAnotherInstance() function, 776
checksum, IP (Internet Protocol), 843
child processes, UNIX processes, 560-563
cihroot jails, 80
cipher block chaining (CBC) mode cipher, 42
circular linked lists, 322
clarity, software design, 32
Clarke, Arthur C., 3
class diagrams, UML (Unified Markup Language), 53
classes
analyzing, CC (code comprehension), 116-117
IP addresses, 832
vulnerabilities
  design vulnerabilities, 14-15
  implementation vulnerabilities, 15-16
  operational vulnerabilities, 16
vulnerabilities, 14
cleanup() function, 792
cleanup_exit() function, 793
Cleaton, Nick, 538
client IP addresses, maintaining state with, 1029-1030
client tier (Web applications), 1042
clients
  client control, 1047-1048
  pipe squatting, 705
  visibility, 1046-1047
close() function, 556-557
close-on-exec file descriptor, UNIX, 581-582
CloseHandle() function, 628
closing
  files, studio file system, 556-557
  TCP connections, 871-872
Clowes, Shaun, 1104
CLR (Common Language Runtime), 6
CLSIDs, mapping to applications, COM (Component Object Model), 728
code
  auditing, 111, 133, 147
  binary navigation tools, 155-157
  CC (code comprehension) strategies, 112-119
  CP (candidate point) strategies, 112, 119-128
  debuggers, 151-154
  dependency analysis, 135-136
  desk checking, 137-139
  DG (design generalization) strategies, 112, 128-133
  fuzz testing tools, 157-158
  internal flow analysis, 133-135
  OpenSSH case study, 158-164
  rereading code, 136-137
  scorecard, 112
  source code navigators, 148-151
  subsystem analysis, 135-136
  test cases, 139-147
  memory, finding in, 188-189
  reuse, 52
  source code profiling, 52
typos, C programming language, 289-296
code access security (CAS). See CAS (code access security), 6
code navigation, 109
  external flow sensitivity, 109-110
  tracing, 111
code page assumptions, Unicode, 455-456
Code Page Mismatch Example listing (8-31), 455
code paths, 135
code review, 92-93
  application review phase, 93, 97-98, 103-105
  bottom-up approach, 100
  hybrid approach, 100-101
  iterative process, 98-99
  peer reviews, 106
  planning, 101-103
  reevaluation, 105
  status checks, 105
  top-down approach, 99
  working papers, 103-104
code auditing, 111, 133, 147
  binary navigation tools, 155-157
  CC (code comprehension) strategies, 112-119
  CP (candidate point) strategies, 112, 119-128
  debuggers, 151-154
  dependency analysis, 135-136
  desk checking, 137-139
  DG (design generalization) strategies, 112, 128-133
  fuzz testing tools, 157-158
  internal flow analysis, 133-135
  OpenSSH case study, 158-164
  rereading code, 136-137
  scorecard, 112
  source code navigators, 148-151
  subsystem analysis, 135-136
  test cases, 139-147
code navigation, 109
  external flow sensitivity, 109-110
  tracing, 111
documentation and analysis phase, 93, 106-108
  findings summary, 106
  preassessment phase, 93
  application access, 95-96
  information collection, 96
  scoping, 94
  process outline, 93
  remediation support phase, 93, 108-109
Code Surfer, 150
code-auditing situations, 9
INDEX

CoInitializeEx() function, 729
ColdFusion, 75
ColdFusion Markup Language (CFML), 1013
ColdFusion MX, 1014
collecttimeout() function, 799
collisions, Windows NT object namespaces, 630-631
COM (Component Object Model), Windows NT
  access controls, 734-736
  Active X security, 749-754
  application audits, 741-749
  application identity, 728, 732-733
  application registration, 741-743
  ATL (Active Template Library), 740
  automation objects, 729, 749
  CLSID mapping, 728
  components, 725-727
  DCOM Configuration utility, 731-732
  impersonation, 736-737
  interface audits, 743-749
  interfaces, 727-728
  IPC (interprocess communications), 725-754
  MIDL (Microsoft Interface Definition Language), 738-740
  OLE (Object Linking and Embedding), 728
  proxies, 730
  stubs, 731
  subsystem access permissions, 733-734
  threading, 729-730
  type libraries, 731
COMbust tool, 749
Common Gateway Interface. See CGI (Common Gateway Interface)
Common Language Runtime (CLR), 6
common real types, 238
Communications of the ACM, 450
Comparison Vulnerability Example listing (6-20), 266
comparisons, type conversions, C programming language, 265-270
compensating controls, operational vulnerabilities, 76
component diagrams, UML (Unified Markup Language), 54
Component Object Model (COM). See COM (Component Object Model)
Computer Security: Art and Science, 5
concurrent programming
  APCs (asynchronous procedure calls), 765
deadlocks, 760-762
  multithreaded programs, 810-825
  process synchronization, 762
    interprocess synchronization, 770-783
    lock matching, 781-783
    synchronization object scoreboard, 780-781
    System V synchronization, 762-764
    Windows NT synchronization, 765-770
  race conditions, 759-760
  reentrancy, 757-759
repetition, 806-809
shared memory segments, 763
signals, 783
  asynchronous-safe function, 791-797, 800-801, 804-809
default actions, 784-785
  handling, 786-788
  interruptions, 791-796, 806-809
  jump locations, 788-791
  non-returning signal handlers, 797-801, 804, 806
  sending, 786
  signal handler scoreboard, 809-810
  signal masks, 785
  vulnerabilities, 791-801, 804-809
starvation, 760
threads
deadlocks, 823-825
  PThreads API, 811-813
  race conditions, 816-823
  starvation, 823-825
Windows API, 813-815
condition variables, PThreads API, 812-813
conditions, ACC logs, unanticipated conditions, 364-365
confidentiality, 41
  encryption
    algorithms, 41-42
    block ciphers, 42
    common vulnerabilities, 43-45
    exchange algorithms, 43
    IV (initialization vector), 42
    stream ciphers, 42
  expectations of, 7-8
configuration files
  OpenSSH, 160
  UNIX, 508-509
configuration settings
  ASP, 1118
  ASP.NET, 1121-1123
  Java servlets, 1112-1113
  PHP, 1104-1105
CONNECT method, 1021
Connection header field (HTTP), 1018
connection points, objects, 736
connections
RPCs (Remote Procedure Calls), 706
TCP (Transmission Control Protocol), 865, 869
  blind connection spoofing, 876-879
  connection tampering, 879
  establishing, 871-872
  fabrication, 875-876
  flags, 870
  resetting, 872
  states, 869-870
ConnectNamedPipe() function, 704
constraint establishment, test cases, code audits, 144-145
Content-Encoding header field (HTTP), 1019
Content-Language header field (HTTP), 1019
Content-Length header field (HTTP), 1019
Content-Location header field (HTTP), 1019
Content-MD5 header field (HTTP), 1019
Content-Range header field (HTTP), 1019
Content-Transfer-Encoding header field (HTTP), 1019
Content-Type header field (HTTP), 1019
CONTENT_LENGTH (environment variable), 1088
CONTENT_TYPE (environment variable), 1088
context handles, RPCs (Remote Procedure Calls), 718-721
contexts, Windows NT sessions, access tokens, 644-645
control flow, auditing, 326-339
flow transfer statements, 336
looping constructs, 327-336
switch statements, 337-339
control-flow sensitive code navigation, 109-110
Controller component (MVC), 1045
controlling terminals, UNIX, 574
conversion rules, type conversions, C programming language, 225-231
ConvertSidToStringSid() function, 637
ConvertStringSidToSid() function, 637
cookies, 1036-1038
stack cookies, 190-191
COPY method, 1022
core files, 519
CoRegisterClassObject() function, 744
Correct Use of GetFullPathName() listing (8-13), 416
corruption (memory), 167
buffer overflows, 168-169
global overflows, 186
heap overflows, 183-186
off-by-one errors, 180-183
process memory layout, 169
SHE (structured exception handling) attacks, 178-180
stack overflows, 169-178
static overflows, 186
protection mechanisms, 189-190
ASLR (address space layout randomization), 194
assessing, 196-202
function pointer obfuscation, 195-196
heap hardening, 191-193
nonexecutable stack, 193
SafeSEH, 194-195
stack cookies, 190-191
shellcode, 187-189
Cost header field (HTTP), 1019
counter (CTR) mode cipher, 42
CP (candidate point), code audits, 112, 119-128
application-specific CPs, 128
automated source analysis tools, 120-122
black box generated CPs, 123-128
general approach, 119-120
simple binary CPs, 122
simple lexical CPs, 122
crackaddr() function, 303
CRC (cyclic redundancy check) routines, 46
Create()() functions, 631
CreateEvent() function, 768
CreateFile() function, 632, 661, 664-665, 667, 674-675, 699-700
CreateHardLink() function, 676
CreateMutex() function, 630, 766
CreateNamedPipe() function, 699-700, 704
CreateNewKey() function, 684
CreatePrivateNamespace() function, 631
CreateProcess() function, 426, 654
CreateRestrictedToken() function, 642
CreateSemaphore() function, 768
CreateWaitableTimer() function, 769
credentials, authorization, untrustworthy credentials, 37
critical sections, Windows API, 814
cross-site scripting
ASP, 1118
ASP.NET, 1121
Java servlets, 1110-1111
Perl, 1096
PHP, 1103
XSS, 1071-1074
cryogenic sleep attacks, 545-546
crypto subsystem, SSH server, code audits, 160
cryptographic hash functions, 46
cryptographic signatures, 47
cryptography, 41
cryptographic data integrity, 45
cryptographic signatures, 47
hash functions, 45-46
originator validation, 47
salt values, 46
cryptography
algorithms, 41-42
block ciphers, 42
common vulnerabilities, 43-45
exchange algorithms, 43
IV (initialization vector), 42
stream ciphers, 42
CRYPTO_realloc_clean() function, 380
Cscope source code navigator, 149
Ctags source code navigator, 149-150
CTR (counter) mode cipher, 42
Cutler, David, 626
cyclic redundancy check (CRC) routines, 46
DACL (discretionary access control list), 632
daemons, UNIX, 467-468
Dangerous Data Type Use listing (7-41), 374
Dangerous Use of IsDBCSLeadByte() listing (8-30), 454
Dangerous Use of strncpy() listing (8-2), 396
data buffers, OpenSSH, vulnerabilities, 307-310
data flow, vulnerabilities, 18-19
data flow diagrams (DFDs), 55-58
data hiding, 307
data integrity, 45
cryptographic signature, 47
hash functions, 45-46
originator validation, 47
salt values, 46
data link layer, network segmentation, 84-85
data ranges, lists, 324, 326
data storage, C programming language, 204-211
data tier (Web applications), 1043-1044
Data Truncation Vulnerability listing (8-11), 415
Data Truncation Vulnerability 2 listing (8-12), 415
data types, application protocols, matching, 927-934
data verification, application protocols, 935
data-flow sensitivee code navigation, 109-110
datagrams, IP datagrams, 834-836
data_xfer() function, 355
Date header field (HTTP), 1019
DCE (Distributed Computing Environment) RPCs, 618, 706
DCOM (Distributed Component Object Model), 328, 725-754, 829
access controls, 734-736
Active X security, 749-754
application audits, 741-749
application identity, 732-733
application registration, 741-743
ATL (Active Template Library), 740
automation objects, fuzz testing, 749
DCOM Configuration utility, 731-732
impersonation, 736-737
interface audits, 743-749
MIDL (Microsoft Interface Definition Language), 738-740
subsystem access permissions, 733-734
DCOM Configuration utility, 731-732
DDE (Dynamic Data Exchange), 658
Windows messaging, 697
DDE Management Library (DDEML) API, 697
de Weger, Benne, 48
deadlocks
concurrent programming, 760, 762
threading, 823-825
debuggers, code auditing, 151-154
DecodePointer() function, 195
DecodeSystemPointer() function, 195
decoding, Unicode, 449-450
Decoding Incorrect Byte Values listing (8-28), 443
decoding routines, RPCs (Remote Procedure Calls), UNIX, 622-623
decomposition, software design, 27-28
default argument promotions, 232, 237
default settings, insecure defaults, 69
default site installations, Web-based applications, 75
Default Switch Case Omission Vulnerability listing (7-24), 338
default type conversions, 224
defense in depth, 31
definition files, RPCs (Remote Procedure Calls), UNIX, 619-622
DELETE method, 1020
delete payloads, ISAKMP (Internet Security Association and Key Management Protocol), 969-971
delete_session() function, 201
Delivering Signals for Fun and Profit, 806
demilitarized zones (DMZs), 86
denial-of-service (DoS) attacks. See DoS (denial of service) attacks
dependency analysis, code audits, 135-136
DER (Distinguished Encoding Rules), ASN.1 (Abstract Syntax Notation), 977-979
Derived-From header field (HTTP), 1019
descriptors, UNIX files, 512-513
design
SDLC (Systems Development Life Cycle), 13
software, 26
abstraction, 27
accuracy, 32
algorithms, 26-27
clarity, 32
decomposition, 27-28
failure handling, 35-36
loose coupling, 33
strong cohesion, 33
strong coupling exploitation, 34
threat modeling, 49-66
transitive trust exploitation, 35
trust relationships, 28-31
vulnerabilities, 14-15
design conformity checks, DG (design generalization) strategy, 131-133
desk checking, code audits, 137-139
desktop object, IPC (interprocess communications), 690-691
Detect_attack Small Packet Algorithm in SSH listing (6-18), 261
Detect_attack Truncation Vulnerability in SSH listing (6-19), 262
developer documentation, reviewing, 51
developers, interviewing, 51
development protective measures, operational vulnerabilities, 76-79
  ASLR (address space layout randomization), 78
  heap protection, 77-78
  nonexecutable stacks, 76
  registered function pointers, 78
  stack protection, 77
  VMs (virtual machines), 79
device files
  UNIX, 511
  Windows NT, 666-668
DeviceIoControl() function, 677
DFDs (data flow diagrams), 55-58
DG (design generalization) strategies, code audits, 112, 128-133
  design conformity check, 131-133
  hypothesis testing, 130-131
  system models, 129-130
Different Behavior of vsnprintf() on Windows and UNIX listing (8-1), 394
Digital Encryption Standard (DES) encryption, 44
Digital Equipment Corporation (DEC) Virtual Memory System (VMS), 626
dilimiters
  embedded delimiters, metacharacters, 408-411
  extraneous dilimiters, 598-601
direct program invocation, UNIX, 565-570
directionality, stateful firewalls, 906
directories, UNIX, 462-464, 514-516
  creating, 500-503
  entries, 514
  Filesystem Hierarchy Standard, 463
  mount points, 463
  parent directories, 503
  permissions, 498-499
  public directories, 507-508
  race conditions, 535-538
  root directories, 574
  safety, 503
  working directories, 574
directory cleaners, UNIX temporary files, 546-547
directory indexing, Web servers, 74
Directory Traversal Vulnerability listing (8-15), 420
discretionary access control list (DACL), 632
Distributed Component Object Model (DCOM). See DCOM (Distributed Component Object Model)
DCE (Distributed Computing Environment) RPCs, 618, 706
Division Vulnerability Example listing (6-27), 274
DllGetClassObject() function, 749
DLLs (dynamic link libraries), 70
  loading, 656-658
  redirection, 657
dlopen() function, 607-608
DMZs (demilitarized zones), 86
dNS (Domain Name System), 984, 989-990
  headers, 991-992
  length variables, 996, 998-1000, 1002
  name servers, 986-987
  names, 993-996
  packets, 991
  question structure, 992
  request traffic, 989
  resource records, 984-985, 993
    conventions, 988
  spoofing, 1002-1005
  zones, 986-987
documentation
  application protocols, collecting, 922-923
  threat modeling, 62-65
documentation phase, code review, 93, 106-108
  findings summary, 106
domain name caches, 986
  Domain Name System (DNS). See DNS (Domain Name System), 984
domain names, 985
domain sockets, UNIX, 615, 617-618
domains, 985
d  error domains, 378-379
DoS (denial-of-service) attacks, 48
  name validation, 931-932
DOS 8.3 filenames, 673-674
doubling Vulnerability listing (7-46), 380
  Double-Free Vulnerability listing (7-45), 379
doubles, auditing, 379-385
doubly linked lists, 322
Dowd, Mark, 895, 967
do_cleanup() function, 793
do_ip() function, 838
do_mremap() function, 342-343
Dragomirescu, Razvan, 1095
DREAD risk ratings, 63-64
Dubee, Nicholas, 478
duplicate elements, 323
dynamic content, 1009
dynamic Data Exchange (DDE). See DDE (Dynamic Data Exchange)
dynamic link libraries (DLLs). See DLLs (dynamic link libraries)
Embedded Delimiter Example listing (8-8), 409
embedded delimiters, metacharacters, 408-411
embedded path information (HTTP), 1022-1023
embedding state in HTML and URLs, 1032-1033
Empty List Vulnerabilities listing (7-12), 322
empty lists, vulnerabilities, 322-323
encapsulation, packets, 920
EncodePointer() function, 195
EncodeSystemPointer() function, 195
encoding
entities, 443
HTML encoding, 443-444
multiple encoding layers, 444-445
parameters, 1026
UTF-16 encoding, 449
UTF-8 encoding, 447-448
XML encoding, 443-444
encryption, 41, 1058-1059
algorithms, 41-42
asymmetric encryption, 42
block ciphers, 42
common vulnerabilities, 43-45
Digital Encryption Standard (DES) encryption, 44
ISAKMP (Internet Security Association and Key Management Protocol), vulnerabilities, 971-972
IV (initialization vector), 42
key exchange algorithms, 43
stream ciphers, 42
symmetric encryption, 41
end user license agreements (EULAs), 9
endpoint mappers, 706
endpoints, RPC servers, binding to, 712-714
enforcing policies, 36-49
enhanced kernel protections, 82
enterprise firewalls, layer 7 inspection, 894
entities (encoded data), 443
entries, UNIX directories, 514
entry points, 50
ENV environment variable (UNIX), 605-606
environment arrays, UNIX file descriptors, 591-611
environment strings, Linux, 594
environment subsystems, 627
environment variables, 1087-1093
PATH_INFO, 1022
UNIX, 603-609
environmental attacks, 21-22
equality operators, 243
err() function, 425
error checking branches, code paths, 135
error domains, 378-379
error messages, overly verbose error messages,
Web-based applications, 75
errors
lists, pointer updates, 323-324
loops, 335-336
escape_sql() function, 434
escaping metacharacters, 439-440
ESP (extended stack pointer), 170
Esser, Stefan, 1103
establishing TCP connections, 871
ETag header field (HTTP), 1019
/etc directory (UNIX), 463
EULAs (end user license agreements), 9
eval() function
Perl, 1095-1096
PHP, 1101-1103
evasion, metacharacter evasion, 441-445
event objects, Windows NT, 767
Example of Bad Counting with Structure Padding listing (6-34), 286
Example of Dangerous Program Use listing (8-19), 428
Example of Structure Padding Double Free listing (6-33), 286
exception handling (EH), C++, 179
exceptional conditions, 22
execl() function, 569
execve() function, ASP, 1117-1118
ExpandEnvironmentStrings() function, 418
Expect header field (HTTP), 1019
expectations, security, 7-9
Expert C Programming, 204
Expires header field (HTTP), 1019
explicit allow filters (white lists), metacharacters, 435-436
explicit deny filters (black lists), metacharacters, 435-436
explicit type conversions, 224
exploiting transitive trusts, 35
Exploiting Software, 168
export function tables, 52
extended base pointer (ESP), 173
extended stack pointer (ESP), 170
Extensible Stylesheet Language Transformations (XSLT), 65, 1012
extensions, UNIX privileges, 491-494
external application invocation, OpenSSH, 161
external entities, 50
external flow sensitivity, code navigation, 109-110
external trust levels, 50
external trusted sources, spoofing attacks, firewalls, 914-915
extraneous delimiters, 598-601
extraneous filename characters, Windows NT, 670-672
extraneous input thinning, test cases, code audits, 145-146
packet-filtering firewalls, 893-896
proxy firewalls, 893-896
spoofing attacks, 914, 919
close spoofing, 917-919
distant spoofing, 914-917
encapsulation, 920
source routing, 920
stateful firewalls, 905
directionality, 906
fragmentation, 907-909
stateful inspection firewalls, 909-913
TCP (Transport Control Protocol), 905-906
UDP (User Datagram Protocol), 906
stateless firewalls, 896
fragmentation, 902-905
FTP (File Transfer Protocol), 901
TCP (Transmission Control Protocol), 896-898
UDP (User Datagram Protocol), 899-901
flags
ACEs, 650
TCP connections, 870
URG flags, TCP (Transmission Control Protocol), 889-890
floating points, conversions, 230-231
floating types, C programming language, 205
floats, 225
flow, control flow, auditing, 326-339
flow analysis, 111
flow transfer statements, auditing, 336
fopen() function, 548-549
fork() function, 560-561, 563-565
format specifiers, 422-423
Format String Vulnerability in a Logging Routine listing (8-17), 424
Format String Vulnerability in WU-FTPD listing (8-16), 423
format strings, 422-425
formats, metacharacters, 418
format strings, 422-425
path metacharacters, 418-422
Perl open() function, 429-431
shell metacharacters, 425-429
SQL queries, 431-434
forms (HTTP), 1024-1025
forward() method, Java servlets, 1108
forward-tracing code, 111
fprintf() function, 422
fragmentation
IP (Internet Protocol), 853-863
overlapping fragments, 858-862
pathological fragment sets, 855-858
processing, 854-855
stateful firewalls, 907-909
stateless firewalls, 902-905
zero-length fragments, 909
Frasunek, Przemysław, 500
fread() function, 550-551, 553-555
free() function, 315, 379-385, 792
FreeBSD, 460
privileges, dropping temporarily, 487-488
From header field (HTTP), 1019
fscanf() function, 554
fstat() function, 528-531
ftok() function, 777
FTP (File Transfer Protocol), 899-901, 922
active FTP, 900
passive FTP, 901
stateless firewalls, 901
fully functional resolvers (DNS), 986
function pointers
obfuscation, 195-196
registration of, 78
Function Prologue listing (5-1), 174
function prototypes, C programming language, type conversions, 232
functions
access(), 527
AdjustTokenGroups(), 643
AdjustTokenPrivileges(), 643
alloc(), 318
allocation functions, auditing, 369-377
apr_palloc(), 299
auditing, 339
argument meaning, 360-362
audit logs, 339-340
return value testing, 340-350
side-effects, 351-359
authenticate(), 177
bounded string functions, 393-395, 397-400
BUF_MEM_grow() function, 311-312
calling conventions, 173
checkForAnotherInstance(), 776
cleanup(), 792
cleanup_exit(), 793
close(), 556-557
CloseHandle(), 628
ColInitializeEx(), 729
collecttimeout(), 799
ConnectNamedPipe(), 704
ConvertSidTo(stringsid, 637
ConvertStringSidTo()int, 637
CoRegisterClassObject(), 744
crackaddr(), 303
Create*(), 631
CreateEvent(), 768
CreateFile(), 632, 661, 664-665, 667, 674-675
CreateHardLink(), 676
CreateMutex(), 630, 766
CreateNamedPipe(), 699-700, 704
CreateNewKey(), 684

INDEX
INDEX

CreatePrivateNamespace( ), 631
CreateProcess( ), 426, 654
CreateRestrictedToken( ), 642
CreateSemaphore( ), 768
CreateWaitableTimer( ), 769
CRYPTO_realloc_clean( ), 380
data_xfer( ), 355
DecodePointer( ), 195
DecodeSystemPointer( ), 195
delte_session( ), 201
DeviceIoControl( ), 677
DllGetClassObject( ), 749
dlopen( ), 607-608
do_cleanup( ), 793
do_ip( ), 838
do_mremap( ), 342-343
do_mremap( ), 342-343
do_mremap( ), 342-343
do_mremap( ), 342-343
do_mremap( ), 342-343
do_mremap( ), 342-343
do_mremap( ), 342-343
do_mremap( ), 342-343
do_mremap( ), 342-343
do_mremap( ), 342-343
do_mremap( ), 342-343
do_mremap( ), 342-343
do_mremap( ), 342-343
do_mremap( ), 342-343
DecrementEnvironmentStrings( ), 418
decrement_environmentstrings( ), 418
decrement_environmentstrings( ), 418
decrement_environmentstrings( ), 418
decrement_environmentstrings( ), 418
decrement_environmentstrings( ), 418
decrement_environmentstrings( ), 418
decrement_environmentstrings( ), 418
decrement_environmentstrings( ), 418
decrement_environmentstrings( ), 418
decrement_environmentstrings( ), 418
decrement_environmentstrings( ), 418
execute( ), 187, 426, 566-567, 591-592
ExpandEnvironmentStrings( ), 418
close( ), 557
cntl( ), 586
gets( ), 551, 553
open( ), 548-549
fork(), 560-561, 563-565
fprintf( ), 422
fread( ), 550-551, 553-555
free( ), 315, 379-385, 792
fscanf( ), 554
fstat( ), 528-531
ftok( ), 777
function_A( ), 171
function_B( ), 172
GetCurrentProcess( ), 632
GetFullPathName( ), 416-418
GetLastError( ), 631, 778
GetMachineName( ), 328
getlimit( ), 574
get_mac( ), 201
get_string_from_network( ), 371
get_user( ), 181
ImpersonateNamedPipe( ), 700-703
initgroups( ), 477
initialize_ipc( ), 777
initJobThreads( ), 773
input_userauth_info_response( ), 216
invocations, C programming language, 237-238
IsDBCSLeadByte( ), 454
kill( ), 786
list_add( ), 757
list_init( ), 757
longjump( ), 788-791
lreplly( ), 423
lstat( ), 528-531
make_table( ), 216
malloc( ), 341, 371
memset( ), 199
mktemp( ), 543
mkstemp( ), 542-543
mktemp( ), 539, 541
MultiByteToWideChar( ), 451-452, 456-457
my_malloc( ), 371
NitQuerySystemInformation( ), 632
open( ), 429-431, 563-565
OpenFile( ), 661
OpenMutex( ), 630
OpenPrivateNamespace( ), 631
OpenProcess( ), 632
parent functions, vulnerabilities, 318
parse_rrecord( ), 998
php_error_docref( ), 332
pipe( ), 612
pop( ), 170
popen( ), 426, 429-431
pcre( ), 526, 535
printf( ), 425, 555-556
processJob( ), 773
processNetwork( ), 773
processThread( ), 783
process_file( ), 792
process_login( ), 385
process_string( ), 181
process_tcp_packet( ), 841
process_token_string( ), 353
push( ), 170
putenv( ), 596-598
pw_lock( ), 585
QueryInterface( ), 744-747
read( ), 315
read_data( ), 314
read_line( ), 358
realloc( ), 341-342
reentrancy, 757-759
RegCloseKey( ), 628
RegCreateKey( ), 420
RegCreateKeyEx( ), 420, 683
RegDeleteKey( ), 421
RegDeleteKeyEx( ), 421
RegDeleteValue( ), 421
RegOpenKey( ), 422
RegOpenKeyEx( ), 422
RegQueryValue( ), 420
RegQueryValueEx( ), 420
retrieve_data( ), 758
return values
  finding, 344
  ignoring, 341-346
  misinterpreting, 346-350
INDEX

rfork(), 562
RpcBindingInqAuthClient(), 715
RpcServerListen(), 714
RpcServerRegisterAuthInfo(), 715
RpcServerRegisterIf(), 711-712
RpcServerRegisterIfEx(), 711-712
RpcServerUseProtseq(), 712
RpcServerUseProtseqEx(), 713
SAPI_POST_READER_FUNC(), 332
scanf(), 388-389
search_orders(), 434
semget(), 777
setegid(), 476
setenv(), 576-577, 596-598
seteuid(), 468-471
setgid(), 476
setgroups(), 477
setjump(), 788-791
setregid(), 476
setreuid(), 472-473
setresgid(), 476
setresuid(), 476
setrlimit(), 574
SetThreadToken(), 647
setuid(), 468, 470-472
ShellExecute(), 655
ShellExecuteEx(), 655
stdio.h, 327
side-effects
referentially opaque side effects, 351
referentially transparent side effects, 351
siglongjmp(), 788-791
signal(), 786-788, 807
sigsetjmp(), 788-791
sizeof(), 181, 272
snprintf(), 394-395, 414, 416
socketpair(), 615, 617-618
sprintf(), 177, 389-391, 414
stat(), 527-531
strcat(), 392-393
strncpy(), 391-392, 400
strcat(), 399-400
strlen(), 181
strcat(), 397-398
strncpy(), 395, 397
syslog(), 425
system(), 426
tempnam(), 541-542
TerminateThread(), 782
tgetent(), 609
time(), 1005
tmpfile(), 543
tmpnam(), 541-542
toupper(), 255
ttryLib(), 578
unbounded string functions, 388-393

Unicode, 450-457
UNIX
  group ID functions, 475-477
  user ID functions, 468-475
unlink(), 535-537, 618
uselib(), 578
utility functions, HTTP (Hypertext Transfer Protocol), 941-942
vfork(), 562
vreply(), 424
vsnprintf(), 424
wait functions, 765
wcscpy(), 453
WideCharToMultiByte(), 451-452, 457
_xwprintfW(), 391
_xlate_ascii_write(), 354
function_A() function, 171
function_B(), 172
function_B() function, 172
fuzz testing
  automation objects, COM (Component Object Model), 749
code auditing tools, 157-158

G
gates, Bill, 625
gateways, 834
  system call gateways, 82
GATEWAY_INTERFACE (environment variable), 1088
GECOS field, UNIX, 462
general CP (candidate point) strategy, code audits, 119-120
generalization approach, application review, 100
GET method, 1023, 1026
GetCurrentProcess() function, 632
GetFullPathName() Call in Apache 2.2.0
  listing (8-14), 417
GetFullPathName() function, 416-418
GetLastError() function, 631, 778
GetMachineName() function, 328
getrlimit() function, 574
get_mac() function, 201
get_string_from_network() function, 371
global namespaces, Windows NT, 629
global overflows, 186
globbing characters, UNIX programs, indirect invocation, 571
GNU/Linux, 460
Govindavajhala, Sudhakar, 80
Greenman, David, 801
group ID functions (UNIX), 475-477
group IDs (GIDs), UNIX, 461
global overflows, 186
globbing characters, UNIX programs, indirect invocation, 571
GNU/Linux, 460
Govindavajhala, Sudhakar, 80
Greenman, David, 801
group ID functions (UNIX), 475-477
group IDs (GIDs), UNIX, 461
global overflows, 186
INDEX

group lists, Windows NT sessions, SIDs, 641
groups, UNIX, 461-462
  effective groups, 465
  file security, 494-512
  GIDs (group IDs), 461, 465, 475-477
  login groups, 461
  primary groups, 461
  privilege vulnerabilities, 477-494
  process groups, 609-611
  real groups, 465
  saved set groups, 465
  secondary groups, 461
  setgid (set-group-id), 464
  supplemental groups, 461, 465
Guninski, Giorgi, 588, 591

H

Hacker Emergency Response Team (HERT), 877
handlers, non-returning signal handlers, signals, 797-801, 804-806
handles, Windows NT objects, 632-636
handling  
signals, 786-788
strings, C programming language, 388-407
hard links  
UNIX files, 515, 522-525
Windows NT files, 676
hardware device drivers, 511
Hart, Johnson M., 654
hash functions, 43-46
hash payloads, ISAKMP (Internet Security Association and Key Management Protocol), 964-965
hash tables, auditing, 321-322, 326
hash-based message authentication code (HMAC), 47
hashing algorithms, 326
headers  
DNS (Domain Name System), 991-992
HTTP (Hypertext Transport Protocol), 1018-1020
  fields, 1018-1020
  parsing, 937-938
IP (Internet Protocol), validation, 836-844
ISAKMP (Internet Security Association and Key Management Protocol), 949-952
certificate payloads, 963-964
delete payloads, 969-971
hash payloads, 964-965
identification payloads, 961-963
key exchange payloads, 959, 961
nonce payloads, 965-966
notification payloads, 966-968
proposals payloads, 956-958
security association payloads, 956
signature payloads, 965
transform payloads, 959
vendor ID payloads, 971
TCP headers, 865
  validation, 866-867
UDP headers, validation, 864
headers (HTTP), Referer, 1030-1031
heap hardening, 191-193
heap overflows, buffer overflows, 183-186
heap protection, operational vulnerabilities, preventing, 77-78
Henriksen, Inge, 1089
HERT (Hacker Emergency Response Team), 877
Hex-encoded Pathname Vulnerability listing (8-27), 441
hexadecimal encoding, pathnames, vulnerabilities, 441-443
hidden fields, auditing, 1036
high-level attack vectors, OpenSSH, code auditing, 162-164
HKEY_CLASSES_ROOT key, 726
HMAC (hash-based message authentication code), 47
Hoglund, Greg, 168
home directories, UNIX users, 462
/home directory (UNIX), 463
HOME environment variable (UNIX), 604-605
homographic attacks, 1060
  Unicode, 450
Host header field (HTTP), 1019
host-based firewalls, 82
host-based IDSs (intrusion detection systems), 83
host-based IPSs (intrusion prevention systems), 83
host-based measures, operational vulnerabilities, 79-83
  antimalware applications, 82-83
  change monitoring, 83
  chroot jails, 80
  enhanced kernel protections, 82
  file system permissions, 79
  host-based firewalls, 82
  host-based IDSs (intrusion detection systems), 83
  host-based IPSs (intrusion prevention systems), 83
  object system permissions, 79
  restricted accounts, 80
  system virtualization, 81
How to Survive a Robot Uprising, xvii
Howard, Michael, 50, 647-648, 736
HPUX, 460
HTML (Hypertext Markup Language), 1009
  encoding, 443-444
HTTP (Hypertext Transport Protocol), 921, 937-948, 1009
  authentication, 1033-1036, 1056-1057
  cookies, 1036-1038
  embedded path information, 1022-1023
  forms, 1024-1025

1145
headers, 1018-1020
fields, 1018-1020
parsing, 937-938
methods, 1020
CONNECT, 1021
DELETE, 1020
GET, 1023, 1026
OPTIONS, 1021
parameter encoding, 1026
POST, 1025-1026
PUT, 1020
SPACEJUMP, 1021
TEXTSEARCH, 1021
TRACE, 1021
WebDAV (Web Distributed Authoring and Versioning) methods, 1022
overview of, 1014
posting data, 942-948
query strings, 1023-1024
requests, 1014-1016, 1030-1031
resource access, 940-941
responses, 1016-1017
sessions, 1038-1039, 1049-1052
security vulnerabilities, 1051-1052
session management, 1052-1053
session tokens, 1053-1056
state maintenance, 1027-1029
client IP addresses, 1029-1030
cookies, 1036-1038
embedding state in HTML and URLs, 1032-1033
HTTP authentication, 1033-1036, 1056-1057
Reffer request headers, 1030-1031
sessions, 1038-1039, 1049, 1051-1056
utility functions, 941-942
versions, 1017-1018
HTTP request methods, 73
hybrid approach, application review, 100-101
Hypertext Markup Language (HTML). See HTML (Hypertext Markup Language)
Hypertext Transfer Protocol (HTTP). See HTTP (Hypertext Transport Protocol)
hypothesis testing, DG (design generalization) strategy, 130-131

IDA Pro binary navigation tool, 156
IDC (Internet Database Connection), 1013
identification payloads, ISAKMP (Internet Security Association and Key Management Protocol), 961-963
idioms, UNIX privileges, misuse of, 486-487
IDL files, RPCs (Remote Procedure Calls), 708-710
IDs, files, UNIX, 494-495
IDSs (intrusion detection systems), 88
host-based IDSs (intrusion detection systems), 83
If Header Processing Vulnerability in Apache’s mod_dav Module listing (8-6), 404
If-Match header field (HTTP), 1019
If-Modified-Since header field (HTTP), 1019
If-None-Match header field (HTTP), 1019
If-Range header field (HTTP), 1019
If-Unmodified-Since header field (HTTP), 1019
Ignoring realloc() Return Value listing (7-25), 341
Ignoring Return Values listing (7-28), 345
ImpersonateNamedPipe() function, 700-703
impersonation, 1059-1060
DCOM (Distributed Component Object Model), 736-737
IPC (interprocess communications), 688-689
levels, 688-689
SelimpersonatePrivilege, 689
RPCs (Remote Procedure Calls), 716-717
Windows NT sessions, access tokens, 647
implementation
SDLC (Systems Development Life Cycle), 13
vulnerabilities, 15-16
implementation analysis, OpenSSH, code auditing, 161-162
implementation defined behavior, C programming language, 204
implicit type conversions, 224
import function tables, 52
imports, Windows binary layout, 70
in-band representation, metadata, 407
in-house software audits, 10
.inc files
ASP, 1118
PHP, 1105
include() method, Java servlets, 1108
Incorrect Temporary Privilege Relinquishment in FreeBSD Inetd listing (9-2), 487
independent research, 10
indexed queries, 1024, 1086-1087
Indirect Memory Corruption listing (5-5), 199
indirect program invocation, UNIX, 570-572
information collection
application review, 96
threat modeling, 50-53
inheritance
ACLs (access control lists), Windows NT, 649
Windows NT object handles, 633-636
initgroups() function, 477
initialization, variables, auditing, 312-315
initialization vector (IV), 42
initialize_ipc() function, 777
initJobThreads() function, 773
INDEX

inline evaluation
  ASP, 1117-1118
  ASP.NET, 1121
  Java servlets, 1110
  Perl, 1095-1096
  PHP, 1101-1103

input
  extraneous input thinning, 145-146
  malicious input, tracing, 113-114
  treating as hostile, 144
  vulnerabilities, 18-19

input_userauth_info_response( ) function, 216

insecure defaults, 69

insufficient validation, authentication, 38

integer conversion rank, 233

integer overflow, 927-928

Integer Overflow Example listing (6-2), 215

Integer Overflow with 0-Byte Allocation Check
  listing (7-37), 370

Integer Sign Boundary Vulnerability Example in
  OpenSSL 0.9.6I listing (6-6), 222

integer types, C programming language, 205-206

integer underflow, 928, 930-931

type promotions, 233-238

signed integers
  boundaries, 220-223
  vulnerabilities, 246-248

type conversions, 228-229

narrowing, 227-228

sign extensions, 248-265

value preservation, 225-226

widening, 226-227

unsigned integers
  boundaries, 213-218, 220
  numeric overflow, 215-217
  numeric underflow, 217-218

vulnerabilities, 246-248

integration, SDLC (Systems Development Life Cycle), 13

integrity, 45

  auditing, importance of, 9, 11
  common vulnerabilities, 47-48
  cryptographic signatures, 47
  expectations of, 8
  hash functions, 45-46
  originator validation, 47
  salt values, 46

Intel architectures
  carry flags (CFs), 214
  multiplication overflows, 218, 220

interface proxies, COM (Component Object Model), 730

interfaces
  COM (Component Object Model) applications, 727-728
  auditing, 743-749
  network interfaces, 832
  RPC servers, registering, 711-712
  vulnerabilities, 21

internal flow analysis, code auditing, 133-135

internal trusted sources, spoofing attacks, firewalls, 915

Internet Database Connection (IDC), 1013

Internet Server Application Programming Interface (ISAPI), 1010

interprocess communication, UNIX, 611-618

interprocess communications (IPC). See IPC
  (interprocess communications)

interprocess synchronization, vulnerabilities, 770-783

interruptions, signals, 791-796, 806-809

interviewing developers, 51

intrusion prevention systems (IPSs). See IPSs
  (intrusion prevention systems)

INVALID_HANDLE_VALUE, NULL, compared, 632-633

invocation
  DCOM objects, 735-736
  UNIX programs, 565-572
  direct invocation, 565-570
  indirect invocation, 570-572

IP (Internet Protocol), 831-832

  addresses, 832-833
    maintaining state with, 1029-1030
    addressing, 833-834
    checksum, 843
    fragmentation, 853-863
    overlapping fragments, 858-862
    pathological fragment sets, 855-858
    processing, 854-855
    header validation, 836-844
    IP packets, 834-836
    options, 844-851
    source routing, 851-853
    subnet, 832

IPC (interprocess communications), Windows NT, 685

  COM (Component Object Model), 725-754
  DDE (Dynamic Data Exchange), 697
  desktop object, 690-691
  mailslots, 705-706
  impersonation, 688-689
  pipes, 698-705
  redirector, 686-688
  RPCs (Remote Procedure Calls), 706-724
  security, 686-689
shatter attacks, 694-697
window station, 690
WTS (Windows Terminal Services), 697-698
IPSs (intrusion prevention systems), 88
host-based IPSs (intrusion prevention systems), 83
IRIX, 460
ISAKMP (Internet Security Association and Key Management Protocol), 948
encryption vulnerabilities, 971-972
headers, 949-952
payloads, 952-956
certificate payloads, 963-964
certificate request payloads, 964
delete payloads, 969-971
hash payloads, 964-965
identification payloads, 961-963
key exchange payloads, 959, 961
nonce payloads, 965-966
notification payloads, 966-968
proposal payloads, 956-958
SA (security association) payloads, 956
signature payloads, 965
transform payloads, 959
vendor ID payloads, 971
ISAPI (Internet Server Application Programming Interface), 1010
ISAPI filters, 71
IsDBCSLeadByte() function, 454
iterative process, application review, 98-99

J

Jaa, Tony, 685
Java Database Connectivity (JDBC), 1106
Java servlets, 1014, 1105-1106
configuration settings, 1112-1113
cross-site scripting, 1110-1111
file access, 1107-1108
file inclusion, 1108-1109
inline evaluation, 1110
JSP file inclusion, 1109-1110
shell invocation, 1108
SQL injection queries, 1106-1107
threading, 1111-1112
Web server APIs versus, 1106
Java Virtual Machine (JVM), 6
JavaScript Object Notation (JSON), 1085
JavaServer Pages (JSP), 1013-1014, 1106
file inclusion, 1109-1110
JDBC (Java Database Connectivity), 1106
Johansson, Eric, 1060
Johnson, Nick, 459
JSON (JavaScript Object Notation), 1085
JSP (JavaServer Pages), 1013, 1106
file inclusion, 1109-1110
jump locations, signals, 788-791
junction points, Windows NT files, 676-680
arbitrary file accesses, 678-680
race conditions, 680
TOCTTOU (time of check to time of use), 680
JVM (Java Virtual Machine), 6

K

kernel
Linux, probing, 569
UNIX, 461
kernel files, UNIX, 511
Kernel Object Manager (KOM), 627
Kernel Probe Vulnerability in Linux 2.2
listing (10-1), 569
key exchange payloads, ISAKMP (Internet Security Association and Key Management Protocol), 959, 961
keys, Windows NT registry
key squatting, 682-684
permissions, 681-682
predefined keys, 681
kill bit, Active X controls, 752
kill() function, 786
Kirch, Olaf, 545
Klima, Vlastimil, 48
KOM (Kernel Object Manager), 627
Kozioi, Jack, 168
Krahmer, Sebastian, 606, 877
Kuhn, Juan Pablo Martinez, 885

L

Lai, Xuejia, 48
languages (programming), C, 203-204
arithmetic boundary conditions, 211-223
binary encoding, 207-208
bit fields, 205
bitwise shift operators, 236-237
byte order, 209
character types, 205
data storage, 204-211
floating types, 205
function invocations, 237-238
implementation defined behavior, 204
integer types, 205-206
macros, 288-289
objects, 205
operators, 271-277
order of evaluation, 282-283
pointers, 277-282
precedence, 287-288
preprocessor, 288-289
signed integer boundaries, 220-223
standards, 204
structure padding, 284-287
INDEX

switch statements, 237

type conversion vulnerabilities, 246-270
type conversions, 223-246
types, 204-207
typos, 289-296
unary operator, 236
unary + operator, 235
unary - operator, 235
undefined behavior, 204
unsigned integer boundaries, 213-218, 220

Last Stage of Delirium (LSD), 188
Last-Modified header field (HTTP), 1019
layer 1 (physical), network segmentation, 84
layer 2 (data link), network segmentation, 84-85
layer 3 (network), network segmentation, 85
layer 4 (transport), network segmentation, 85-87
layer 5 (session), network segmentation, 87
layer 6 (presentation), network segmentation, 87
layer 7 (application)
    enterprise firewalls, 894
    network segmentation, 87-88
laying, static firewalls, 911-913
layers
    multiple encoding layers, 444-445
    network segmentation, 84-87
LD_LIBRARY_PATH environment variable
    (UNIX), 607
LD_PRELOAD environment variable (UNIX), 607
Le Blanc, David, 50
leaks, file descriptors, UNIX, 582-587
Leblanc, David, 235, 647-648, 736
Lebras, Gregory, 1100
Leidl, Bruce, 885
length calculations, multiple calculations on same
    input, 367-369
Length Miscalculation Example for Constructing an
    ACC log listing (7-33), 362
length variables, DNS (Domain Name System), 996,
    998-1000, 1002
Lenstra, Arjen, 48
levels, impersonation, IPC (interprocess
    communications, 688-689
libraries, 499-500
    UNIX, 510
Lincoln, Abraham, 167
linked lists
    auditing, 321-326
circular linked lists, 322
doubly linked lists, 322
    singly linked lists, 322
linking objects, vulnerabilities, 607-608
links
    UNIX files, 515-525
        hard links, 515, 522-525
        soft links, 515-522
    Windows NT files, 676-680
        hard links, 676
        junction points, 676-680
Linux, 459-460
capabilities, 492-494
do_mremap() function, vulnerabilities, 342-343
environment strings, 594
    file system IDs, 491
    kernel probes, vulnerabilities, 569
teardrop vulnerability, 325
Linux do_mremap() Vulnerability listing (7-26), 342
Linux Teardrop Vulnerability listing (7-14), 325
List Pointer Update Error listing (7-13), 324
listings
    5-1 (Function Prologue), 174
    5-2 (Off-by-One Length Miscalculation), 175
    5-3 (Off-by-One Length Miscalculation), 181
    5-4 (Overflowing into Local Variables), 197
    5-5 (Indirect Memory Corruption), 199
    5-6 (Off-by-One Overwrite), 200
    6-1 (Twos Complement Representation
        of -15), 209
    6-2 (Integer Overflow Example), 215
    6-3 (Challenge-Response Integer Overflow
        Example in OpenSSH 3.1), 216
    6-4 (Unsigned Integer Underflow Example), 217
    6-5 (Signed Integer Vulnerability Example), 221
    6-6 (Integer Sign Boundary Vulnerability
        Example in OpenSSL 0.9.6l), 222
    6-7 (Signed Comparison Vulnerability
        Example), 247
    6-8 (Antisniff v1.0 Vulnerability), 250
    6-9 (Antisniff v1.1 Vulnerability), 251
    6-10 (Antisniff v1.1.1 Vulnerability), 252
    6-11 (Antisniff v1.1.2 Vulnerability), 253
    6-12 (Sign Extension Vulnerability Example), 254
    6-13 (Prescan Sign Extension Vulnerability in
        Sendmail), 256
    6-14 (Sign-Extension Example), 258
    6-15 (Zero-Extension Example), 258
    6-16 (Truncation Vulnerability Example in
        NFS), 260
    6-17 (Truncation Vulnerability Example), 260
    6-18 (Detect_attack Small Packet Algorithm in
        SSH), 261
    6-19 (Detect_attack Truncation Vulnerability in
        SSH), 262
    6-20 (Comparison Vulnerability Example), 266
    6-21 (Signed Comparison Vulnerability), 267
    6-22 (Unsigned Comparison Vulnerability), 267
    6-23 (Signed Comparison Example in PHP), 269
    6-24 (Sizeof Misuse Vulnerability Example), 271
    6-25 (Sign-Preserving Right Shift), 273
    6-26 (Right Shift Vulnerability Example), 273
    6-27 (Division Vulnerability Example), 274

1149
INDEX

6-28 (Modulus Vulnerability Example), 275
6-29 (Pointer Arithmetic Vulnerability Example), 281
6-30 (Order of Evaluation Logic Vulnerability), 283
6-31 (Order of Evaluation Macro Vulnerability), 283
6-32 (Structure Padding in a Network Protocol), 284
6-33 (Example of Structure Padding Double Free), 286
6-34 (Example of Bad Counting with Structure Padding), 286
7-1 (Apache mod_dav CDATA Parsing Vulnerability), 298
7-2 (Bind 9.2.1 Resolver Code gethostans() Vulnerability), 300
7-3 (Sendmail crackaddr() Related Variables Vulnerability), 304
7-4 (OpenSSH Buffer Corruption Vulnerability), 307
7-5 (OpenSSL BUF_MEM_grow() Signed Variable Desynchronization), 311
7-6 (Uninitialized Variable Usage), 313
7-7 (Uninitialized Memory Buffer), 314
7-8 (Uninitialized Object Attributes), 314
7-9 (Arithmetic Vulnerability Example), 317
7-10 (Arithmetic Vulnerability Example in the Parent Function), 318
7-11 (Type Confusion), 320
7-12 (Empty List Vulnerabilities), 322
7-13 (List Pointer Update Error), 324
7-14 (Linux Teardrop Vulnerability), 325
7-15 (Simple Nonterminating Buffer Overflow Loop), 328
7-16 (MS-RPC DCOM Buffer Overflow Listing), 329
7-17 (NTPD Buffer Overflow Example), 329
7-18 (Apache mod_php Nonterminating Buffer Vulnerability), 331
7-19 (Apache 1.3.29/2.X mod_rewrite Off-by-one Vulnerability), 332
7-20 (OpenBSD ftp Off-by-one Vulnerability), 333
7-21 (Postincrement Loop Vulnerability), 334
7-22 (Pretest Loop Vulnerability), 335
7-23 (Break Statement Omission Vulnerability), 337
7-24 (Default Switch Case Omission Vulnerability), 338
7-25 (Ignoring realloc() Return Value), 341
7-26 (Linux do_mremap() Vulnerability), 342
7-27 (Finding Return Values), 344
7-28 (Ignoring Return Values), 345
7-29 (Unexpected Return Values), 347
7-30 (Outdated Pointer Vulnerability), 351
7-31 (Outdated Pointer Use in ProFTPD), 354
7-32 (Sendmail Return Value Update Vulnerability), 356
7-33 (Length Miscalculation Example for Constructing an ACC log), 362
7-34 (Buffer Overflow in NSS Library’s ssl2_HandleClientHelloMessage), 365
7-35 (Out-of-Order Statements), 366
7-36 (Netscape NSS Library UCS2 Length Miscalculation), 367
7-37 (Integer Overflow with 0-Byte Allocation Check), 370
7-38 (Allocator-Rounding Vulnerability), 372
7-39 (Allocator with Header Data Structure), 372
7-40 (Reallocation Integer Overflow), 373
7-41 (Dangerous Data Type Use), 374
7-42 (Problems with 64-bit Systems), 375
7-43 (Maximum Limit on Memory Allocation), 376
7-44 (Maximum Memory Allocation Limit Vulnerability), 377
7-45 (Double-Free Vulnerability), 379
7-46 (Double-Free Vulnerability in OpenSSL), 380
7-47 (Reallocation Double-Free Vulnerability), 383
8-1 (Different Behavior of vsnprintf() on Windows and UNIX), 394
8-2 (Dangerous Use of strncpy()), 396
8-3 (Strcpy()-like Loop), 400
8-4 (Character Expansion Buffer Overflow), 401
8-5 (Vulnerable Hex-Decoding Routine for URIs), 404
8-6 (If Header Processing Vulnerability in Apache’s mod_dav Module), 404
8-7 (Text-Processing Error in Apache mod_mime), 406
8-8 (Embedded Delimiter Example), 409
8-9 (Multiple Embedded Delimiters), 410
8-10 (NUL-Byte Injection with Memory Corruption), 413
8-11 (Data Truncation Vulnerability), 415
8-12 (Data Truncation Vulnerability 2), 415
8-13 (Correct Use of GetFullPathName()), 416
8-14 (GetFullPathName() Call in Apache 2.2.0), 417
8-15 (Directory Traversal Vulnerability), 420
8-16 (Format String Vulnerability in WU-FTP), 423
8-17 (Format String Vulnerability in a Logging Routine), 424
8-18 (Shell Metacharacter Injection Vulnerability), 426
8-19 (Example of Dangerous Program Use), 428
8-20 (SQL Injection Vulnerability), 431
8-21 (SQL Truncation Vulnerability), 433
INDEX

8-22 (Character Black-List Filter), 435
8-23 (Character White-List Filter), 436
8-24 (Metacharacter Vulnerability in PCNFSD), 437
8-25 (Vulnerability in Filtering a Character Sequence), 437
8-26 (Vulnerability in Filtering a Character Sequence #2), 438
8-27 (Hex-encoded P pathname Vulnerability), 441
8-28 (Decoding Incorrect Byte Values), 443
8-29 (Return Value Checking of MultiByteToWideChar( )), 452
8-30 (Dangerous Use of IsDBCSLeadByte( )), 454
8-31 (Code Page Mismatch Example), 455
8-32 (NUL Bytes in Multibyte Code Pages), 456
9-1 (Privilege Misuse in XFree86 SVG A Server), 478
9-2 (Incorrect Temporary Privilege Relinquishment in FreeBSD Inetd), 487
9-3 (Race Condition in access( ) and open( )), 526
9-4 (Race Condition from Kerberos 4 in Istat( ) and open( )), 529
9-5 (Race Condition in open( ) and Istat( )), 529
9-6 (Reopening a Temporary File), 542
10-1 (Kernel Probe Vulnerability in Linux 2.2), 569
10-2 (Setenv( ) Vulnerability in BSD), 576
10-3 (Misuse of putenv( ) in Solaris Telnetd), 597
13-1 (Signal Interruption), 791
13-2 (Signal Race Vulnerability in WU-FTPD), 802
13-3 (Race Condition in the Linux Kernel’s Uselib( )), 821
16-1 (Name Validation Denial of Service), 931
16-2 (Certificate Payload Integer Underflow in CheckPoint ISAKMP), 954
lists
  auditing, 321-324, 326
data ranges, 324, 326
duplicate elements, 323
empty lists, vulnerabilities, 322-323
linked lists, 322
pointer updates, errors, 323-324
list_add( ) function, 757
list_init( ) function, 757
little-endian architecture, bytes, ordering, 209
loading
  DLLs, 656-658
  Processes, Windows NT, 654-655
  local namespaces, Windows NT, 629
  local privilege separation socket, OpenSSH, 161
  Location header field (HTTP), 1019
  lock matching, synchronization objects, 781-783
  LOCK method, 1022
  log files, UNIX, 510
logic
  business logic, 1041
  presentation logic, 1040-1041
login groups, UNIX, 461
logon rights, Windows NT sessions, 638
longjmp( ) function, 788-791
looping constructs, auditing, 327-336
loops
  data copy, 330
  posttest loops, 334-335
  pretest loops, 334-335
  terminating conditions, 327-334
typos, 335-336
loose coupling, software design, 33
loosely coupled modules, 33
Lopatic, Thomas, 895, 903, 907-911
lreply( ) function, 423
LSD (Last Stage of Delirium), 188
Istat( ) function, 528-531
M
%m format specifier, 423
MAC (Media Address Control), 84
Macros, C programming language, 288-289
magic_quotes option (PHP), 1105
mail spools, UNIX, 509
mailslot squatting, 706
mailslots, Windows NT, IPC (interprocess communications), 705-706
Maimon, Uriel, 897
maintaining state, 1027-1029
  client IP addresses, 1029-1030
  cookies, 1036-1038
  embedding state in HTML and URLs, 1032-1033
  HTTP authentication, 1033-1036, 1056-1057
  Referer request header, 1030-1031
  sessions, 1038-1039, 1049-1052
  security vulnerabilities, 1051-1052
  session management, 1052-1053
  session tokens, 1053-1056
  stateful versus stateless systems, 1027
maintenance, SDLC (Systems Development Life Cycle), 13
major components, 50
make_table( ) function, 216
malicious input, tracing, 113-114
malloc( ) function, 341, 371
man-in-the-middle attacks, 162
management, sessions, 1052-1053
mapping CLSIDs to applications, 728
Max-Forwards header field (HTTP), 1019
Maximum Limit on Memory Allocation listing (7-43), 376
Maximum Memory Allocation Limit Vulnerability listing (7-44), 377
1151
McDonald, John, 571, 903, 907, 911
McGraw, Gary, 168
Media Address Control (MAC), 84
Mehta, Neel, 203, 895, 967
memory, 0 bytes, allocating, 370-371
memory blocks, shared memory blocks, 201-202
memory buffers, uninitialized memory buffers, 314
memory corruption, 167

assessing, 196-202
buffer overflows, 168-169
global overflows, 186
heap overflows, 183-186
off-by-one errors, 180-183
process memory layout, 169
SHE (structured exception handling) attacks, 178-180
stack overflows, 169-178
static overflows, 186

protection mechanisms, 189-190
ASLR (address space layout randomization), 194
function pointer obfuscation, 195-196
heap hardening, 191-193
nonexecutable stack, 193
SafeSEH, 194-195
stack cookies, 190-191
shellcode, 187-189

memory management, auditing, 362
ACC (allocation-check-copy) logs, 362-369
allocation functions, 369-377
allocator scorecards, 377-379
double-frees, 379-385
error domains, 378-379
memory pages, nonexecutable memory pages, 193
memset( ) function, 199
message queues, 614
Message-Id header field (HTTP), 1019
messaging, Windows NT, IPC (interprocess communications), 689-698
metacharacter evasion, 441-445

 Metacharacter Vulnerability in PCNFS
listing (8-24), 437
metacharacters, 387, 407-408
embedded delimiters, 408-411
filtering, 434-445
character stripping vulnerabilities, 437-439
escaping metacharacters, 439-440
insufficient filtering, 436-437
metacharacter evasion, 441-445
format strings, 422-425
formats, 418
NUL-byte injection, 411-414
path metacharacters, 418-422
file canonicalization, 419-420
Windows registry, 420-422
Perl open( ) function, 429-431
shell metacharacters, 425-429

SQL queries, 431-434
truncation, 414-418
UNIX programs, indirect invocation, 570-571
metadata, 407

methods
CONNECT, 1021
COPY, 1022
DELETE, 1020
GET, 1023, 1026
LOCK, 1022
MKCOL, 1022
MOVE, 1022
OPTIONS, 1021
POST, 1025-1026
PROPFIND, 1022
PROPPATCH, 1022
PUT, 1020
SEARCH, 1022
SPACEJUMP, 1021
TEXTSEARCH, 1021
TRACE, 1021
UNLOCK, 1022
Microsoft Developer Network (MSDN), 626
MIDL (Microsoft Interface Definition Language)
DCOM (Distributed Component Object Model), 738-740
RPCs (Remote Procedure Calls), 708
misinterpreting return values, 346-350
Misuse of putenv( ) in Solaris Telnetd
listing (10-3), 597
mitigating factors, operational vulnerabilities, 76
mitigation, threats, 61
MKCOL method, 1022
mkdtemp( ) function, 543
mkstemp( ) function, 542-543
mktemp( ) function, 539, 541
Model component (MVC), 1045
Model-View-Controller (MVC), 1044-1045
modular arithmetic, 214

modules
analyzing, CC (code comprehension), 114-116
loosely coupled modules, 33
strongly coupled modules, 33

Modulus Vulnerability Example listing (6-28), 275
mount points, UNIX, 463
MOVE method, 1022
MS-RPC DCOM Buffer Overflow Listing
listing (7-16), 329
MSDN (Microsoft Developer Network), 626
MTA (multi-threaded apartment), COM (Component Object Model), 729
multibyte character sequences, interpretation, 455
MultiByteToWideChar( ) function, 451-452, 456-457
Multics (Multiplexed Information and Computing Service), 460

Multiple Embedded Delimiters listing (8-9), 410
INDEX

multiple encoding layers, 444-445
multiple-input test cases, code audits, 143
Multiplexed Information and Computing Service (Multics), 460
multiplication overflows, Intel architectures, 218, 220
multiplicative operators, 243
multithreaded apartment (MTA), COM (Component Object Model), 729
multithreaded programs, synchronization, 810-825
deadlocks, 823-825
PTHreads API, 811-813
race conditions, 816-823
starvation, 823-825
Windows API, 813-815
Murray, Bill, 25
mutex, 756
mutex objects, Windows NT, 766
mutexes, PThreads API, 811-812
MVC (Model-View-Controller), 1044-1045
my_malloc( ) function, 371

N
N-tier architectures, 1041, 1043
business tier, 1042-1044
client tier, 1042
data tier, 1042-1043
MVC (Model-View-Controller), 1044-1045
Web tier, 1042-1045
name servers, DNS (Domain Name System), 986-987
name squatting, 630
Name Validation Denial of Service listing (16-1), 931
named pipes
UNIX, 511
Windows NT, 698-699
names, DNS (Domain Name System), 993-996
namespaces (Windows NT)
global namespaces, 629
local namespaces, 629
objects, 629-632
collisions, 630-631
Vista object namespaces, 631
narrowing integer types, 227-228
NAT (Network Address Translation), 88
National Institute for Standards and Technology (NIST), 44
navigating code, 109
external flow sensitivity, 109-110
tracing, 111
NCACN (network computing architecture connection-oriented protocol), RPCs (Remote Procedure Calls), 707
NCALRPC (network computing architecture local remote procedure call protocol), RPCs (Remote Procedure Calls), 708
NCDAG (network computing architecture datagram protocol), RPCs (Remote Procedure Calls), 707
.NET Common Language Runtime (CLR), 6
NetBSD, 460
netmasks, 833
Netscape NSS Library UCS2 Length Miscalculation listing (7-36), 367
Netscape Server Application Programming Interface (NSAPI), 1010
Network Address Translation (NAT). See NAT (Network Address Translation)
network application protocols, 921
ASN.1 (Abstract Syntax Notation), 972-974
BER (Basic Encoding Rules), 975-979
CER (Canonical Encoding Rules), 976-979
DER (Distinguished Encoding Rules), 977-979
PER (Packed Encoding Rules), 979-983
XER (XML Encoding Rules), 983-984
auditing, 922-937
data type matching, 927-934
data verification, 935
documentation collection, 922-923
identifying elements, 923-927
system resource access, 935-937
DNS (Domain Name System), 984, 989-990
headers, 991-992
length variables, 996-1002
name servers, 986-987
names, 993-996
packets, 991
question structure, 992
request traffic, 989
resolvers, 986-987
resource records, 984-985, 988, 993
spoofing, 1002-1005
zones, 986-987
HTTP (Hypertext Transfer Protocol), 937-948
header parsing, 937-938
posting data, 942-948
resource access, 940-941
utility functions, 941-942
ISAKMP (Internet Security Association and Key Management Protocol), 948
cryptographic vulnerabilities, 971-972
headers, 949-952
payloads, 952-971
network computing architecture connection-oriented protocol (NCACN), RPCs (Remote Procedure Calls), 707
network computing architecture datagram protocol (NCDAG), RPCs (Remote Procedure Calls), 707
network computing architecture local remote procedure call protocol (NCALRPC), RPCs (Remote Procedure Calls), 708
Network File System (NFS), 35
network interfaces, 832
network layer, network segmentation, 85
INDEX

network profiles, vulnerabilities, 73
network protocols, 829-831
  IP (Internet Protocol), 831-832
  addressing, 832-834
  checksum, 843
  fragmentation, 853-863
  header validation, 836-844
  IP packets, 834-836
  options, 844-851
  source routing, 851-853
network application protocols, 921
  ASN.1 (Abstract Syntax Notation), 972-984
  auditing, 922-937
  DNS (Domain Name System), 984-1005
  HTTP (Hypertext Transfer Protocol), 937-948
  ISAKMP (Internet Security Association and Key Management Protocol), 948-972
  TCP (Transmission Control Protocol), 864-866
  connections, 865, 869-872
  header validation, 866-867
  headers, 865
  options, 867-869
  processing, 880-890
  segments, 865
  streams, 865, 872-880
  TCP/IP, 830
  UDP (User Datagram Protocol), 863-864
network segmentation, 84-88
  layer 1 (physical), 84
  layer 2 (data link), 84-85
  layer 3 (network), 85
  layer 4 (transport), 85-87
  layer 5 (session), 87
  layer 6 (presentation), 87
  layer 7 (application), 87-88
network time protocol (NTP) daemon, 329
network-based measures, operational vulnerabilities, 83-89
  NAT (Network Address Translation), 88
  network IDSS, 88
  network IPSs, 88
  segmentation, 84-88
  VPNs (virtual private networks), 88
NFS (Network File System), 35, 73
Nietzsche, Friedrich, 755
NIST (National Institute for Standards and Technology), 44
node types, 60
  non-returning signal handlers, signals, 797-801, 804-806
nonce payloads, ISAKMP (Internet Security Association and Key Management Protocol), 965-966
nonexecutable memory pages, 193
nonexecutable stacks
  heap protection, 193
  operational vulnerabilities, preventing, 76
nonrecursive name servers (DNS), 986
nonroot setgid programs (UNIX), 466
nonroot setuid programs (UNIX), 466
nonsecurable objects, Windows NT, 629
nonsuperuser elevated privileges, UNIX, dropping permanently, 482, 484
Nordell, Mike, 677
notification payloads, ISAKMP (Internet Security Association and Key Management Protocol), 966-968
NSAPI (Netscape Server Application Programming Interface), 1010
NTP (network time protocol) daemon, 329
NTPD Buffer Overflow Example listing (7-17), 329
NtQuerySystemInformation() function, 632
NUL byte injection queries, Perl, 1094
NUL Bytes in Multibyte Code Pages listing (8-32), 456
NUL-Byte injection, 411-414
NUL-Byte Injection with Memory Corruption listing (8-10), 413
NUL-termination, Unicode, 452-453
NULL, INVALID_HANDLE_VALUE, compared, 632-633
null bytes, 1068
numeric overflow, unsigned integers, 215-217
numeric overflow conditions, C programming language, 211-212
numeric underflow, unsigned integers, 217-218
numeric underflow conditions, C programming language, 212
numeric wrapping, C programming language, 212

O
Object Management Group (OMG), 53
  object systems, permissions, 79
objects
  analyzing, CC (code comprehension), 116-117
  C programming language, 205
  change monitoring, 83
  COM (Component Object Model), automation objects, 729, 749
  connection points, 736
  DCOM objects
    activation, 734
    invocation, 735-736
  linking, vulnerabilities, 607-608
  uninitialized attributes, 314-315
  variables, management, 307-312
  Windows NT, 627-629
    boundary descriptor objects, 631
    handles, 632-636
    namespaces, 629-632
    nonsecurable objects, 629
    SymbolicLink objects, 629
    system objects, 628
Oechslin, Philippe, 46
off-by-one errors, buffer overflows, 180-183
Off-by-One Length Miscalculation listing (5-2), 175
Off-by-One Length Miscalculation listing (5-3), 181
Off-by-One Overwrite listing (5-6), 200
OLE (Object Linking and Embedding), COM (Component Object Model), 728
Olsson, Mikael, 911
OMG (Object Management Group), 53
omissions, file descriptors, UNIX, 587-591
ONC (Open Network Computing) RPCs, 618, 706
open() function, 429-431, 563-565
open() system call (UNIX), 501
OpenBSD 2.8, 333, 460
OpenBSD ftp Off-by-one Vulnerability listing (7-20), 333
OpenFile() function, 661
open files, stdio file system, 548-549
OpenMutex() function, 630
OpenPrivateNamespace() function, 631
OpenProcess() function, 632
OpenSSH, 158
authentication files, 161
code auditing, case study, 158-164
configuration file, 160
data buffers, vulnerabilities, 307-310
external application invocation, 161
local privilege separation socket, 161
remote client socket, 161
OpenSSH Buffer Corruption Vulnerability listing (7-4), 307
OpenSSL
BUFFMEM_grow() function, 311-312
double-free vulnerability, 380-382
OpenSSL BUFFMEM_grow() Signed Variable Desynchronization listing (7-5), 311
operands, order of evaluation, 282-283
operating systems, file system interaction, 1066
execution, 1067
file uploading, 1068-1069
null bytes, 1068
path traversal, 1067-1068
programmatic SSI, 1068
operational vulnerabilities, 76
access control, 69-70
attack surfaces, 68
development protective measures, 76-79
ASLR (address space layout randomization), 78
heap protection, 77-78
nonexecutable stacks, 76
registered function pointers, 78-79
stack protection, 77
VMs (virtual machines), 79
exposure, 68-73
host-based measures, 79-83
anti-malware applications, 82-83
change monitoring, 83
chroot jails, 80
enhanced kernel protections, 82
file system permissions, 79
host-based firewalls, 82
host-based IDSs (intrusion detection systems), 83
host-based IPSs (intrusion prevention systems), 83
object system permissions, 79
restricted accounts, 80
system virtualization, 81
insecure defaults, 69
network profiles, 73
network-based measures, 83-89
NAT (Network Address Translation), 88
network IDSs, 88
network IPSs, 88
segmentation, 84-88
VPNs (virtual private networks), 88
secure channels, 71-72
spoofing, 72
unnecessary services, 70-71
Web-specific vulnerabilities
authentication, 75
default site installations, 75
directory indexing, 74
file handlers, 74
HTTP request methods, 73
overly verbose error messages, 75
public-facing administrative interfaces, 76
Web-specific vulnerabilities, 73-76
operational vulnerabilities, 16, 67-68
operations, SDLC (Systems Development Life Cycle), 13
operators
assignment operators, type conversions, 231-232
binary bitwise operators, 243
bitwise shift operators, 236-237
C programming language, 233, 271-277
equality operators, 243
multiplicative operators, 243
question mark operators, 243
relational operators, 243
vulnerabilities
right shift, 272-277
size, 271-272
options
IP (Internet Protocol), 844-851
TCP options, processing, 867-869
OPTIONS method, 1021
order of action, ACC logs, 366-367
order of evaluation, operands, 282-283
Order of Evaluation Logic Vulnerability listing (6-30), 283
Order of Evaluation Macro Vulnerability listing (6-31), 283
originator validation, 47
Osborne, Anthony, 571
out-band representation, metadata, 407
out-of-order statements, 366-367
Out-of-Order Statements listing (7-35), 366
Outdated Pointer Use in ProFTPD listing (7-31), 354
Outdated Pointer Vulnerability listing (7-30), 351
outdated pointers, 351-353
ProFTPD, 354-355
overflow
multiplication overflows, Intel architectures, 218, 220
unsigned integers, 215-217
Overflowing into Local Variables listing (5-4), 197
overlapping fragments, IP (Internet Protocol), 858-862
overly verbose error messages, Web-based applications, 75
overwriting bytes, 198-199
ownership, UNIX files, race conditions, 534
O_CREAT | O_EXCL flag (UNIX), 544
O_EXCL flag (UNIX), 501
P
packets
DNS (Domain Name System), 991
capsulation, 920
IP packets, 834-836
packet sniffers, 923-924
source routing, 920
TCP packets, scanning, 897-898
padding bits, unsigned integer types, 206
page flow, 1048-1049
Paget, Chris, 34
parameterized queries, 1062-1063
parameters, transmitting to Web applications, 1022
embedded path information, 1022-1023
forms, 1024-1025
GET method, 1023, 1026
parameter encoding, 1026
POST method, 1025-1026
query strings, 1023-1024
parent directories, UNIX, 503
parent functions, vulnerabilities, 318
parroted request variables, 1089
parse_recrod() function, 998
parsing HTTP headers, 937-938
passive FTP, 901
password files, UNIX, 461
PATH environment variable (UNIX), 603-604
path information (HTTP), 1022-1023
path metacharacters, 418-422
file canonicalization, 419-420
Windows registry, 420-422
path traversal, 1067-1068
pathnames
hexadecimal encoding, 441-443
UNIX, 462
pathological code paths, 135
pathological fragment sets, IP (Internet Protocol), 855-858
paths
files, UNIX, 503-507
path traversal, 1067-1068
PATH_INFO environment variable, 1022, 1090-1093
PATH_TRANSLATED environment variable, 1090-1093
Payloads, ISAKMP (Internet Security Association and Key Management Protocol), 952-956
certificate payloads, 963-964
certificate request payloads, 964
delete payloads, 969-971
hash payloads, 964-965
identification payloads, 961-963
key exchange payloads, 959, 961
nonce payloads, 965-966
notification payloads, 966-968
proposal payloads, 956-958
SA (security association) payloads, 956
signature payloads, 965
transform payloads, 959
vendor ID payloads, 971
PCI (Payment Card Industry) 1.0 Data Security Requirement, 45
peer reviews, application review, 106
PER (Packed Encoding Rules), ASN.1 (Abstract Syntax Notation), 979-983
Perl, 1093
cross-site scripting, 1096
file access, 1094
file inclusion, 1095
inline evaluation, 1095-1096
open() function, 429-431
shell invocation, 1095
SQL injection queries, 1093-1094
taint mode, 1096
permission bitmasks, 495-497
permissions
  DCOM (Distributed Component Object Model), subsystem access permissions, 733-734
  Directories, UNIX, 498-499
  file access, Windows NT, 659, 661
  file systems, 79
  files, UNIX, 495-497
  mailslots, 705
  object systems, 79
  registry keys, Windows NT, 681-682
  UNIX files, race conditions, 533-534
  Windows NT pipes, 698-699
personal user files, UNIX, 509
phishing, 1059-1060
PHP (PHP Hypertext Preprocessor), 1013, 1096-1097
  configuration settings, 1104-1105
  cross-site scripting, 1103
  file access, 1098-1099
  file inclusion, 1101
  inline evaluation, 1101-1103
  shell invocation, 1099, 1101
  SQL injection queries, 1097-1098
php_error_docref() function, 332
phrack magazine, 168
physical layer, network segmentation, 84
PIDs (process IDs), UNIX, 464
pipe squatting, Windows NT, 703-705
piped() system call, 612
pipes
  UNIX, 612, named pipes, 612-614
  Windows NT
    anonymous pipes, 698
    creating, 699-700
    impersonation, 700-703
    IPC (interprocess communications), 698-705
    named pipes, 698-699
    permissions, 698-699
    pipe squatting, 703-705
PKI (Public Key Infrastructure), 43
point-of-sale (PoS) system, 49
Pointer Arithmetic Vulnerability Example listing (6-29), 281
pointer updates, lists, errors, 323-324
pointers, 225
  arithmetic, 278-280
  C programming language, 277-282
  EBP (extended base pointer), 173
  ESP (extended stack pointer), 170
  function pointers, obfuscation, 195-196
  outdated pointers, 351, 353
  ProFTPD, 354-355
  text strings, incrementing incorrectly, 401-406
  vulnerabilities, 280-282
Pol, Joost, 588
policies (security), 5-7
  access control policy, 38
  breaches, 132
  enforcing, 36-49
pop() function, 170
popen() function, 426, 429-431
Portable Operating System Interface for UNIX (POSIX), 627
PoS (point-of-sale) system, 49
positive decimal integers, binary notation, converting to, 207
positive numbers, decimal conversion from binary notation, 207
POSIX (Portable Operating System Interface for UNIX), 460, 627
  signals, handling, 784
POST method, 1025-1026
Postincrement Loop Vulnerability listing (7-21), 334
posting data, HTTP (Hypertext Transfer Protocol), 942, 944-946, 948
posttest loops, pretest loops, compared, 334-335
Practical Cryptography, 41
Pragma header field (HTTP), 1019
preassessment phase, code review, 93
  application access, 95-96
  information collection, 96
  scoping, 94
precedence, C programming language, 287-288
precision, integer types, 206
predefined registry keys, Windows NT, 681
prepared statements, 1062
preprocessors, C programming language, 288-289
Prescan Sign Extension Vulnerability in Sendmail listing (6-13), 256
prescan() function, 256, 356
presentation layer, network segmentation, 87
  presentation logic, 1040-1041
preshared keys (PSKs), discovery of, 972
Pretest Loop Vulnerability listing (7-22), 335
pretest loops, posttest loops, compared, 334-335
primary groups, UNIX, 461
printf() function, 425, 555-556
Privilege Misuse in XFree86 SVGA Server listing (9-1), 478
privilege separation, SSH server, code audits, 160
  privileges, 28
  UNIX, 464-465
    capabilities, 492-494
    directory permissions, 498-499
    dropping permanently, 479-486, 489
    dropping temporarily, 486-490
    extensions, 491-494
    file IDs, 494-495
    file permissions, 495-497
INDEX

TCP/IP, 830
UDP (User Datagram Protocol), 863-864
REST (Representational State Transfer), 1085
SOAP (Simple Object Access Protocol), 1085
SSL/TLS (Secure Sockets Layer/Transport Layer Security), 1058-1059
text-based protocols, data type matching, 932-934
proxies, COM (Component Object Model), 730
proxy firewalls, 895-896
packet-filtering firewalls, compared, 893-894
Proxy-Authorization header field (HTTP), 1019
pseudo-objects, Windows NT, 629
PSKs (preshared keys), discovery of, 972
PThreads API, 811-813
condition variables, 812-813
mutexes, 811-812
public directories, UNIX, 507-508
Public header field (HTTP), 1019
public key encryption, 42
Public Key Infrastructure (PKI), 43
public-facing administrative interfaces, Web-based applications, 76
punctuation errors, loops, 335-336
punycode, 1060
Purczynski, Wojciech, 494
push() function, 170
PUT method, 1020
putenv() function, 594, 596-598
pw_lock() function, 585

Q
QA testing, 118
queries
indexed queries, 1024
parameterized queries, 1062-1063
query strings, 1023-1024
SQL queries, metacharacters, 431-434
query strings
HTTP, 1023-1024
indexed queries, 1086-1087
QueryInterface() function, 744-747
QUERY_STRING (environment variable), 1091-1093
question mark operators, 243
question structure, DNS (Domain Name System), 992
queues, message queues, 614

R
Race Condition from Kerberos 4 in lstat() and open() listing (9-4), 529
Race Condition in access() and open() listing (9-3), 526
Race Condition in open() and lstat() listing (9-5), 529
Race Condition in the Linux Kernel’s Uselib() listing (13-3), 821
race conditions
  junction points, 680
  synchronicity, 759-760
threading, 816-817, 819, 821-823
UNIX file system, 526-538
directory races, 535-538
ownership races, 534
permission races, 533-534
TOCTOU (time to check to time of use), 527-531
Rain Forest Puppy (RFP), 1094
Range header field (HTTP), 1019
raw memory devices, 511
raw sockets, 467
Raymond, Eric, 541
RDBMS (relational database management system), 431
read() function, 315
reading files, stdio file system, 550-555
read_data() function, 314
read_line() function, 358
real groups, UNIX, 465
real users (UNIX), 464, 574
realloc() function, 341-342
Reallocation Double-Free Vulnerability listing (7-47), 383
Reallocation Integer Overflow listing (7-40), 373
recursive name servers (DNS), 986
redirector, Windows NT, 686-688
session credentials, 687
SMB relay attacks, 688
UNC (Universal Naming Convention) paths, 686
redundancy in Web applications, 1040
reentrancy
  functions, 757-759
  multithreaded programs, 810
referentially opaque side effects, functions, 351
referentially transparent side effects, functions, 351
Referer header field (HTTP), 1019
Referer request header, 1030-1031
RegCloseKey() function, 628
RegCreateKey() function, 420
RegCreateKeyEx() function, 420, 683
RegDeleteKey() function, 421
RegDeleteKeyEx() function, 421
RegDeleteValue() function, 421
registered function pointers, operational vulnerabilities, preventing, 78
registering interfaces, RPC servers, 711-712
register_globals option (PHP), 1104-1105
registration, COM (Component Object Model) applications, 741-743
registry, Windows NT, 680
  key permissions, 681-682
  key squatting, 682-684
  predefined keys, 681
  value squatting, 682-684
RegOpenKey(), 420
RegOpenKey() function, 422
RegOpenKeyEx(), 420
RegOpenKeyEx() function, 422
RegQueryValue() function, 420
RegQueryValueEx() function, 420
relational database management system (RDBMS), 431
relational operators, 243
relationships, variables, 298-307
relinquishing UNIX privileges
  permanently, 479-486, 489
  temporarily, 486-490
remediation support phase, code review, 93, 108-109
remote client socket, OpenSSH, 161
Remote Procedure Call (RPC) endpoints, 50
REMOTE_ADDR (environment variable), 1088
REMOTE_HOST (environment variable), 1088
REMOTE_IDENT (environment variable), 1088
REMOTE_USER (environment variable), 1088
Reopening a Temporary File listing (9-6), 542
reply generation, 806-809
Represenational State Transfer (REST), 1085
request traffic, DNS (Domain Name System), 989
request variables, 1088
  parroted request variables, 1089
  synthesized request variables, 1089-1091
requests
  HTTP, 1014-1016
  Referer request header, 1030-1031
  RPC servers, listening to, 714
REQUEST_METHOD (environment variable), 1088
require() function, 1095
requirements, software, 15
requirements definitions, SDLC (Systems Development Life Cycle), 13
rereading code, code audits, 136-137
resetting TCP connections, 872
resolvers, DNS (Domain Name System), 986-987
resource limits, UNIX, 574-580
resource records, DNS (Domain Name System), 984-985, 993
  conventions, 988
responses (HTTP), 1016-1017
  spoofing for, 916
REST (Representational State Transfer), 1085
restricted accounts, operational vulnerabilities, preventing, 80
restricted tokens, Windows NT sessions, access tokens, 642-644
retention, process attributes, UNIX, 573-574
retrieve_data() function, 758
Retry-After header field (HTTP), 1019
Return Value Checking of MultiByteToWideChar() listing (8-29), 452
return value testing, functions, 340-350
INDEX

return values, functions
  finding, 344
  ignoring, 341-346
  misinterpreting, 346-350
reuse
code, 52
  UNIX temporary files, 544-546
reverse-engineering applications, 924-927
reviewing code, 92-93
  application review phase, 91-93, 97-98, 103-105
  bottom-up approach, 100
  hybrid approach, 100-101
  iterative process, 98-99
  peer reviews, 106
  planning, 101-103
  reevaluation, 105
  status checks, 105
  top-down approach, 99
  working papers, 103-104
code auditing, 111, 133, 147
  binary navigation tools, 155-157
  CC (code comprehension) strategies, 112-117, 119
  CP (candidate point) strategies, 112, 119-120, 122-128
dependency analysis, 135-136
desk checking, 137-139
  DG (design generalization) strategies, 112, 128-133
design generalization
fuzz testing tools, 157-158
  internal flow analysis, 133-135
  OpenSSH case study, 158-164
  rereading code, 136-137
  scorecard, 112
  source code navigators, 148-151
  subsystem analysis, 135-136
test cases, 139-140, 142-147
code navigation, 109
testing, 111
documentation and analysis phase, 93, 106-108
  findings summary, 106
  preassessment phase, 93
  application access, 95-96
  information collection, 96
  scoping, 94
  process outline, 93
  remediation support phase, 93, 108-109
  Rey, Enno, 972
rfork() function, 562
RPC (Remote Procedure Calls) servers, 711-716
  authentication, 714-716
  endpoints, 50
    binding to, 712-714
  interfaces, registering, 711-712
  requests, listening to, 714
  RpcBindingInqAuthClient() function, 715
  RPCs (Remote Procedure Calls)
    UNIX, 618-624
      authentication, 623-624
      decoding routines, 622-623
      definition files, 619-622
  Windows NT
    ACFs (application configuration files), 710
    application audits, 722-724
    connections, 706
decoding routines, 714
  context handles, 718-721
  DCE (Distributed Computing Environment)
    RPCs, 706
    IDL file structure, 708-710
    impersonation, 716-717
    IPC (interprocess communications), 706-724
    MIDL (Microsoft Interface Definition Language), 708
    ONC (Open Network Computing) RPCs, 706
    proprietary state mechanisms, 721
  RPC servers, 711-716
  threading, 721
  transports, 707-708
  RpcServerListen() function, 714
  RpcServerRegisterAuthInfo() function, 715
  RpcServerRegisterIf() function, 711-712
  RpcServerRegisterIfEx() function, 711-712
  RpcServerUseProtseq() function, 712
  RpcServerUseProtseqEx() function, 713
  running code, auditing, 567
  runtime stack, activation records, 170
  Russinovich, Mark E., 628-629, 654

S

  Sacerdoti, David, 567
  SAFER (Software Restriction Policies) API,
    Windows NT sessions, access tokens, 644
  SafeSEH, 194-195
  salt values, 46
  sandboxing, 53
  SAPI_POST_READER_FUNC() function, 332
  saved set groups (UNIX), 465
  saved set users (UNIX), 465
  saved set-user-IDs (UNIX), 574
  saved-set-group-IDs (UNIX), 574
  sa_handler, 788
  /sbin directory (UNIX), 463
  scanf() functions, 388-389

1161
INDEX

scanning, 53
TCP packets, 897-898
Schneier, Bruce, 41
SCM (Services Control Manager), 658-659
SCO, 460
scoping, code review, 94
scorecards, code audits, 112
script URI, 1089
scripts
server-side scripting, 1013-1014
SCRIPT_NAME (environment variable), 1091-1093
SDLC (Systems Development Life Cycle), code audits, 13
SEARCH method, 1022
search_orders( ) function, 434
second order injection, 1064-1065
second-order injection attacks, 409
secondary groups, UNIX, 461
securable objects, Windows NT, 628
secure channels, 71-72
Secure Programming, 647
Secure Sockets Layer/Transport Layer Security (SSL/TLS), 87, 1058-1059
Secure Sockets Layer (SSL). See SSL (Secure Sockets Layer)
securelevels (BSD), 492
security
access control, 1057-1058
C/C++ problems, 1075
expectations, 7-9
OS and file system interaction, 1066
execution, 1067
file uploading, 1068-1069
null bytes, 1068
path traversal, 1067-1068
programmatic SSL, 1068
phishing and impersonation, 1059-1060
policies, enforcing, 36-49
SQL injection, 1061-1062
parameterized queries, 1062-1063
prepared statements, 1062
second order injection, 1064-1065
stored procedures, 1063-1064
testing for, 1065-1066
threading issues, 1074
Web environments, 1075-1078
XML injection, 1069-1070
XPath injection, 1070-1071
XSS (cross-site scripting), 1071-1074
security association (SA) payloads, ISAKMP (Internet Security Association and Key Management Protocol), 956
security breaches, policy breaches, compared, 132
security descriptors, Windows NT, 647-648
access masks, 648-649
ACL inheritance, 649
ACL permissions, 652-653
programming interfaces, 649-652
strings, 651-652
segmentation (network), 84-88
layer 1 (physical), 84
layer 2 (data link), 84-85
layer 3 (network), 85
layer 4 (transport), 85-87
layer 5 (session), 87
layer 6 (presentation), 87
layer 7 (application), 87-88
segments, TCP (Transmission Control Protocol), 865
SEH (structured exception handling) attacks, 178-180, 194-195
SelimpersonatePrivilege, IPC (interprocess communications), 689
semaphore sets, 614
semaphores
System V IPC, 763-764
Windows NT, 768
semget() function, 777
sending signals, 786
Sendmail
crackaddr() function, vulnerabilities, 303
prescan sign extension vulnerability, 256-257
return values, update vulnerability, 356
Sendmail crackaddr() Related Variables Vulnerability listing (7-3), 304
Sendmail Return Value Update Vulnerability listing (7-32), 356
sentinel nodes, 323
sequence numbers, TCP (Transmission Control Protocol), 884-888
Server header field (HTTP), 1019
Server Message Blocks (SMBs), 73, 688
server-side includes (SSIs), 1011
server-side scripting, 1013-1014
server-side transformation, 1012
servers
automation servers, 729
name servers, DNS (Domain Name System), 986-987
pipe squatting, 704-705
Web servers
APIs, 1010-1011
server-side scripting, 1013-1014
server-side transformation, 1012
SSIs (server-side includes), 1011
SERVER_NAME (environment variable), 1089-1090
SERVER_PORT (environment variable), 1090
SERVER_PROTOCOL (environment variable), 1090
SERVER_SOFTWARE (environment variable), 1088

1162
INDEX

service image paths, 659
service-oriented architecture (SOA), 1084
services, Windows NT, 658-659
servlets. See Java servlets
session credentials, redirector, 687
session layer, network segmentation, 87
session tokens, 1039, 1053-1056
sessions
HTTP, 1038-1039, 1049-1052
security vulnerabilities, 1051-1052
session management, 1052-1053
session tokens, 1053-1056
UNIX, process sessions, 609-611
Windows NT, 636-645, 647
access tokens, 639-645, 647
logon rights, 638
SIDs (security IDs), 637-638
setegid( ) function, 476
setenv( ) function, 576-577, 596-598
Setenv() Vulnerability in BSD listing (10-2), 576
seteuid( ) function, 468-470
setgid (set-group-id), UNIX, 464
setgid programs (UNIX), 466
setgid( ) function, 476
setgroups() function, 477
setjump( ) function, 788-791
setreuid( ) function, 476
setresgid( ) function, 476
setresuid( ) function, 472-473
setreuid() function, 473-475
setrlimit( ) function, 574
SetThreadToken() function, 647
settings, default settings, insecure defaults, 69
setuid (set-user-id), UNIX, 464
setuid programs (UNIX), 466
setuid root programs (UNIX), 466-467
setuid( ) function, 468, 470-472
SGML (Standard Generalized Markup Language), 1009
shadow password files, UNIX, 461
shared key encryption, 41
shared libraries, 499-500
shared memory, multiple processes, 756
shared memory blocks, 201-202
shared memory segments, 614
synchronization, 763
sharing files, UNIX, 564-565
shatter attacks, Windows messaging, 694-697
SHELL environment variable (UNIX), 606
shell environment variables, UNIX, 603
shell histories, UNIX, 509
shell invocation
ASP, 1115
ASP.NET, 1120
Java servlets, 1108
Perl, 1095
PHP, 1099, 1101
shell login scripts, UNIX, 509
shell logout scripts, UNIX, 509
Shell Metacharacter Injection Vulnerability listing (8-18), 426
shell metacharacters, 425-429
shellcode, 178, 187-189
Shellcoder's Handbook, The, 118, 168
ShellExecute() function, 655
ShellExecuteEx() function, 655
shells, UNIX users, 462
side-effects, functions
auditing, 351-359
referentially opaque side effects, 351
referentially transparent side effects, 351
SIDs (security IDs), Windows NT, 637-645, 647
siglongjmp() function, 788-791
sign bit
arithmetic schemes, 207
signed integer types, 206
Sign Extension Vulnerability Example listing (6-12), 254
sign extensions, 226
type conversions, 248-265
truncation, 259-265
Sign-Extension Example listing (6-14), 258
Sign-Preserving Right Shift listing (6-25), 273
signal handler scoreboard, 809-810
Signal Interruption listing (13-1), 791
signal marks, 573
signal masks, 785
Signal Race Vulnerability in WU-FTPD listing (13-2), 802
signal() function, 786-788, 807
signals, 783
asynchronous-safe function, 791-797, 800-801, 804-809
default actions, 784-785
handling, 786-788
interruptions, 791-796, 806-809
jump locations, 788-791
non-returning signal handlers, 797-801, 804-806
repetition, 806-809
sending, 786
signal handler scoreboard, 809-810
signal masks, 785
vulnerabilities, 791-801, 804-809
signature payloads, ISAKMP (Internet Security Association and Key Management Protocol), 965
signatures, cryptographic signatures, 47
Signed Comparison Example in PHP listing (6-23), 269
Signed Comparison Vulnerability Example listing (6-7), 247
Signed Comparison Vulnerability listing (6-21), 267
signed integer types, C programming language, 206
Signed Integer Vulnerability Example listing (6-5), 221

1163
signed integers
  boundaries, 220-223
  conversions, 228-229
  vulnerabilities, 246-248
  narrowing, 227-228
  sign bit, arithmetic schemes, 207
  widening, 226-227
signing Active X controls, 750
sigsetjmp() function, 788-791
SIGSTOP default action, 787
simple binary CPs (candidate points), 122
simple lexical CPs (candidate points), 122
Simple Mail Transfer Protocol (SMTP), 921
Simple Nonterminating Buffer Overflow Loop listing (7-15), 328
Simple Object Access Protocol (SOAP), 1085
simple type conversions, C programming language, 231-232
  single sign-on (SSO) system, 75
  single-threaded apartment (STA), COM (Component Object Model), 729
  singly linked lists, 322
site-restricted controls, Active X, 752
size, operators, vulnerabilities, 271-272
Sizeof Misuse Vulnerability Example listing (6-24), 271
sizeof() function, 181, 272
SMB relay attacks, 688
SMBs (Server Message Blocks), 73, 688
SMTP (Simple Mail Transfer Protocol), 921
sniffing attacks, 162
snort reassembly vulnerability, TCP (Transmission Control Protocol), 885-890
snprintf() function, 394-395, 414, 416
Snyder, Window, 50
SOA (service-oriented architecture), 1084
SOAP (Simple Object Access Protocol), 1085
socketpair() function, 615, 617-618
soft links, UNIX files, 515, 517-522
software, 3
  requirements, 15
  security expectations, 7-9
  specifications, 15
  vulnerabilities, 4-5, 18
    bugs, 4-5
    classifying, 14-17
data flow, 18-19
design vulnerabilities, 14-15
environmental attacks, 21-22
exceptional conditions, 22
implementation vulnerabilities, 15-16
input, 18-19
interfaces, 21
operational vulnerabilities, 16
security policies, 5-7
trust relationships, 19-20
software design, 26
  abstraction, 27
  accuracy, 32
  algorithms, 26-27
  application architecture modeling, 53-66
  clarity, 32
development, 27-28
  failure handling, 35-36
  loose coupling, 33
  strong cohesion, 33
  strong coupling exploitation, 34
  threat modeling, 49-50
  information collection, 50-53
transitive trust exploitation, 35
trust relationships, 28-31
  chain of trust relationships, 30-31
  complex trust boundaries, 30
  defense in depth, 31
  simple trust boundaries, 28-30
Software Restriction Policies (SAFER) API. See SAFER (Software Restriction Policies) API
Solaris, 460
Solomon, David A., 628, 654
Song, Dug, 907
source code, profiling, 52
source code audits, COM (Component Object Model), 743
source code navigators, code audits, 148-151
  Code Surfer, 150
  Cscope, 149
  Ctags, 149-150
  Source Navigator, 150
  Understand, 151
Source Navigator, 150
source routing
  IP (Internet Protocol), 851-853
  packets, 920
source-only application access, 95
SPACEJUMP method, 1021
specialization approach, application review, 99
specifications, software, 15
SPIKE fuzz testing tool, 158
spoofing, 72
  DNS (Domain Name System), 1002-1005
  TCP streams, 874-875
  blind connection spoofing, 876-879
  spoofing attacks, firewalls, 914, 919
  close spoofing, 917-919
  distant spoofing, 914-917
  encapsulation, 920
  source routing, 920
sprintf() functions, 177, 389-391, 414
SQL (Structured Query Language)
  queries, metacharacters, 431-434
  SQL injection, 1061-1062
  ASP, 1113, 1115
  ASP.NET, 1118-1119
pointers
  incorrect increments, 401-406
typos, 406-407
unbounded copies, 400
unbounded string functions, 388-393
Windows NT security descriptors, 651-652
strcat() function, 399-400
strncpy() function, 397-398
strlen() function, 181
strong cohesion, software design, 33
strong coupling, software design exploitation, 34
strongly coupled modules, 33
structure padding, C programming language, 284-287
Structure Padding in a Network Protocol listing (6-32), 284
structured exception handling (SHE) attacks, 178-180
structures, variables, management, 307-312
Struts framework, 1008
stub resolvers (DNS), 986
subdomains, 985
subnet addresses, 832-834
subsystem access permissions, DCOM (Distributed Component Object Model), 733-734
subsystem analysis, code audits, 135-136
superusers, UNIX, 461
supplemental group privileges, UNIX, dropping permanently, 480-481
supplemental groups, UNIX, 461, 465, 574
Swiderski, Frank, 50
switch statements
  auditing, 337-339
  C programming language, 237
switching, 84
symbolic links, UNIX files, 515, 517-522
SymbolicLink objects, 629
symmetric encryption, 41
  block ciphers, 42
synchronization, 756-757
  APCs (asynchronous procedure calls), 765
deadlocks, 760, 762
  multithreaded programs, 810-825
process synchronization, 762
  interprocess synchronization, 770-783
  lock matching, 781-783
  synchronization object scoreboard, 780-781
  System V synchronization, 762-764
  Windows NT synchronization, 765-770
race conditions, 759-760
reentrancy, 757-759
shared memory segments, 763
signals, 783
  asynchronous-safe function, 791-797, 800-801, 804-809
default actions, 784-785
handling, 786-788
interruptions, 791-796, 806-809
jump locations, 788-791
non-returning signal handlers, 797-801, 804-806
repetition, 806-809
sending, 786
signal handler scoreboard, 809-810
signal masks, 785
vulnerabilities, 791-801, 804-809
starvation, 760
threads
deadlocks, 823-825
  PThreads API, 811-813
race conditions, 816-823
starvation, 823-825
  Windows API, 813-815
  synchronization object scoreboard, 780-781
tabulation highlighting, 148
synthesized request variables, 1089-1091
SysInternals, 636
syslog() function, 425
  system call gateways, 82
deadlocks, 823-825
  PThreads API, 811-813
  race conditions, 816-823
  starvation, 823-825
state processing, 880-885
  sequence number boundary condition, 888
  sequence number representation, 884-888
  state processing, 880-885
  URG pointer processing, 889-890
  window scale option, 889
segments, 865
  tables, auditing, 321-322, 326
taint mode, Perl, 1096
  tampering TCP connections, 879
TCP (Transmission Control Protocol), 35, 864-866
  connections, 865, 869
closing, 871-872
establishing, 871
  flags, 870
resetting, 872
states, 869-870
  header validation, 866-867
headers, 865
options, processing, 867-869
processing, 880
  sequence number boundary condition, 888
  sequence number representation, 884-888
  state processing, 880-885
  URG pointer processing, 889-890
  window scale option, 889
tools
code audits, 147
  binary navigation tools, 155-157
debuggers, 151-154
fuzz testing tools, 157-158
OpenSSH case study, 158-164
  source code navigators, 148-151
UNIX, 461
top-down approach, application review, 99
top-down progression, 28
toupper( ) function, 255
TRACe method, 1021
tracing
  black box hits, 117-119
code, 111
  malicious input, 113-114
Trailer header field (HTTP), 1020
transformations, XSLT (Extensible Stylesheet Language Transformation), 1012
Transfer-Encoding header field (HTTP), 1020
transform payloads, ISAKMP (Internet Security Association and Key Management Protocol), 959
transitive trusts, exploiting, 35
Transmission Control Protocol (TCP), 35
  transport layer, network segmentation, 85-87
transports, RPCs (Remote Procedure Calls), 707-708
truncation
  file paths, 415
  integer types, 227-228
  metacharacters, 414-418
  NFS, 260
  sign extensions, 259-265
Truncation Vulnerability Example in NFS listing (6-16), 260
Truncation Vulnerability Example listing (6-17), 260
trust boundaries, 28
  complex trust boundaries, 30
  simple trust boundaries, 28-30
trust domains, 28
trust models, 28
trust relationships
  software design, 28-31
    chain of trust relationships, 30-31
    complex trust boundaries, 30
    defense in depth, 31
    simple trust boundaries, 28-30
  vulnerabilities, 19-20
trusted authorities, 29
trusts, transitive trusts, exploiting, 35
try_lib( ) function, 578
Twos Complement Representation of -15 listing (6-1), 209
type coercions. See type conversions
type confusion, 319, 321
Type Confusion listing (7-11), 320
type conversions, C programming language, 223-248
  assignment operators, 231-232
  comparisons, 265-270
  conversion rules, 225-231
  default type conversions, 224
  explicit type conversions, 224
  floating point types, 230-231
  function prototypes, 232
  implicit type conversions, 224
  integer promotions, 233-238
  narrowing, 227-228
  sign extensions, 248-265
  simple conversions, 231-232
typecasts, 231
  usual arithmetic conversions, 238-245
  value preservation, 225-226
  vulnerabilities, 246-270
  widening, 226-227
type libraries, COM (Component Object Model), 731, 743
typecasts, C programming language, 231
types, C programming language, 204-207
typos
  C programming language, 289-296
  loops, 335-336
  text strings, 406-407
UDP (User Datagram Protocol), 35, 863-864
  header validation, 864
  stateful firewalls, 906
  stateless firewalls, 899-901
UIDs (user IDs), UNIX, 461, 464-465
UML (Unified Markup Language), 53
  class diagrams, 53-54
  component diagrams, 54
  use cases, 54
UN*X, 459
unary operator, C programming language, 236
unary + operator, C programming language, 235
unary – operator, C programming language, 235
unbounded copies, strings, 400
unbounded string functions, 388-393
UNC (Universal Naming Convention), redirector, 686
unconstrained data types, test cases, code audits, 146-147
undefined behavior, C programming language, 204
underflow, unsigned integers, 217-218
Understand source code navigator, 151
Unexpected Return Values listing (7-29), 347
Unicode, 446-447
  character equivalence, 456-457
  code page assumptions, 455-456
  decoding, 449-450
homographic attacks, 450
NUL-termination, 452-453
UTF-16 encoding, 449
UTF-8 encoding, 447-448
Windows functions, 450-457
Unicos, 460
Unified Markup Language (UML). See UML (Unified Markup Language)
Uniform Resource Identifiers (URIs), 1009
Uninformed magazine, 168
Uninitialized Memory Buffer listing (7-7), 314
Uninitialized Object Attributes listing (7-8), 314
Uninitialized Variable Usage listing (7-6), 313
unique creation, UNIX temporary files, 538-544
uninitialized memory buffers, 314
uninitialized object attributes, 314-315
uninitialized variable usage, 313
UNIX, 459
BSD, 459
securelevels, 492
directories, 462-464
creating, 500-503
creating directories, 514
Filesystem Hierarchy Standard, 463
mount points, 463
parent directories, 503
permissions, 498-499
public directories, 507-508
root directories, 574
safety, 503
working directories, 574
domain sockets, 615, 617-618
event variables, 603-609
file descriptors, 580-588, 590-591
file IDs, 494-495
file security, 494-512
files, 462-464, 508, 512
boot files, 511
creating, 500-503
directories, 512-513
device files, 511
directories, 514-516
filenames, 503-507
inodes, 513-514
kernel files, 511
libraries, 510
links, 513-517-525
log files, 510
named pipes, 511
pathnames, 462
paths, 503-507
permissions, 495-497
personal user files, 509
proc file system, 511
program configuration files, 510
program files, 510
race conditions, 526-538
sharing, 564-565
stdio file interface, 547-557
system configuration files, 508-509
temporary files, 538-547
GECOS field, 462
groups, 461-462
effective groups, 465
GIDs, 465
GIDs (group IDs), 461
login groups, 461
primary groups, 461
real groups, 465
saved set groups, 465
secondary groups, 461
setgid (set-group-id), 464
supplementary groups, 461, 465
kernel, 461
Linux, 459
capabilities, 492-494
file system IDs, 491
mail spools, 509
naming of, 460
open() system call, 501
origins of, 459-460
O_EXCL flag, 501
password files, 461
pipes, 612-614
POSIX standards, 460
privileges, 464-465
dropping permanently, 479-486, 489
dropping temporarily, 486-490
extensions, 491-494
group ID functions, 475-477
management code audits, 488-490
programs, 466-468
user ID functions, 468-475
vulnerabilities, 477-494
processes, 464, 560
attributes, 572-611
child processes, 563
children, 560
creating, 560-562
environment arrays, 591-611
fork() system call, 563-565
groups, 609-611
interprocess communication, 611-618
open() function, 563-565
RPCs (Remote Procedure Calls), 618-624
sessions, 609-611
system file table, 563
terminals, 609-611
termination, 562
INDEX

program invocation, 565-572
  direct invocation, 565-570
  indirect invocation, 570-572
  resource limits, 574-580
RPCs (Remote Procedure Calls)
  authentication, 623-624
  decoding routines, 622-623
definition files, 619-622
shadow password files, 461
shell histories, 509
shell login scripts, 509
System V-IPC mechanisms, 614-615
tools, 461
UNIX, 459
users, 461-462
  effective users, 464-465
  home directories, 462
  real users, 464
  saved set users, 465
  setuid (set-user-id), 464
  shells, 462
  superusers, 461
UIDs (user IDs), UNIX, 461
  functions, 468-475
User-Agent header field (HTTP), 1020
users, UNIX, 461-462
  effective users, 464-465
  file security, 494-512
  home directories, 462
  privilege vulnerabilities, 477-494
  real users, 464
  saved set users, 465
  setuid (set-user-id), 464
  shells, 462
  superusers, 461
UIDs (user IDs), 464-465
user ID functions, 468-475
user IDs (UIDs), UNIX, 461
unlink( ) function, 535-537, 618
UNLOCK method, 1022
unmask file permissions, 497
unmask attribute, UNIX, 574
unnecessary services, 70-71
Unsigned Comparison Vulnerability listing (6-22), 267
unsigned integer types, C programming language, 206
Unsigned Integer Underflow Example listing (6-4), 217
unsigned integers
  boundaries, 213-218, 220
  conversions, 228-229
  vulnerabilities, 246-248
  narrowing, 227-228
  numeric overflow, 215-217
  numeric underflow, 217-218
  widening, 226-227
unsigned-preserving promotions, 234
untrustworthy credentials, authentication, 37
Upgrade header field (HTTP), 1020
uploading files, security, 1068-1069
URG flags, TCP (Transmission Control Protocol), 889-890
URI header field (HTTP), 1020
URIs (Uniform Resource Identifiers), 1009
  script URI, 1089
URLs, embedding state in, 1032-1033
use cases, UML (Unified Markup Language), 54
use scenarios, 51
uselib( ) function, 578
User Datagram Protocol (UDP), 35

user IDs (UIDs), UNIX, 461
  functions, 468-475
User-Agent header field (HTTP), 1020
users, UNIX, 461-462
  effective users, 464-465
  file security, 494-512
  home directories, 462
  privilege vulnerabilities, 477-494
  real users, 464
  saved set users, 465
  setuid (set-user-id), 464
  shells, 462
  superusers, 461
UIDs (user IDs), 464-465
user ID functions, 468-475
user IDs (UIDs), UNIX, 461
unlink( ) function, 535-537, 618
UNLOCK method, 1022
unmask file permissions, 497
unmask attribute, UNIX, 574
unnecessary services, 70-71
Unsigned Comparison Vulnerability listing (6-22), 267
unsigned integer types, C programming language, 206
Unsigned Integer Underflow Example listing (6-4), 217
unsigned integers
  boundaries, 213-218, 220
  conversions, 228-229
  vulnerabilities, 246-248
  narrowing, 227-228
  numeric overflow, 215-217
  numeric underflow, 217-218
  widening, 226-227
unsigned-preserving promotions, 234
untrustworthy credentials, authentication, 37
Upgrade header field (HTTP), 1020
uploading files, security, 1068-1069
URG flags, TCP (Transmission Control Protocol), 889-890
URI header field (HTTP), 1020
URIs (Uniform Resource Identifiers), 1009
  script URI, 1089
URLs, embedding state in, 1032-1033
use cases, UML (Unified Markup Language), 54
use scenarios, 51
uselib( ) function, 578
User Datagram Protocol (UDP), 35

validation
  authorization, insufficient validation, 38
  IP headers, 836-844
  name validation, DoS (denial of service) attacks, 931-932
  originator validation, 47
  TCP headers, 866-867
  UDP headers, 864
  value bits, unsigned integer types, 206
  value preservation, C programming language, 225-226
  value-preserving promotions, 234
  values, Windows NT registry, value squatting, 682-684
  Van der Linden, Peter, 204
/var directory (UNIX), 463
variables
  auditing, 298-326
    arithmetic boundaries, 316-319
    initialization, 312-315
    lists, 321-326
    object management, 307-312
    structure management, 307-312
    tables, 321-322, 326
    type confusion, 319, 321
  environment variables, 1087-1093
  PATH_INFO, 1022
  PThread API, condition variables, 812-813
  relationships, 298-303, 305-307
Vary header field (HTTP), 1020
VBScript, 1117-1118
vendor ID payloads, ISAKMP (Internet Security Association and Key Management Protocol), 971
INDEX

Version header field (HTTP), 1020
versions of HTTP (Hypertext Transport Protocol), 1017-1018
vfork() function, 562
Via header field (HTTP), 1020
View component (MVC), 1045
ViewState, ASP.NET, 1121
virtual device drivers, 511
virtual memory areas (VMAs), 343
Virtual Memory System (VMS), 626
virtual private machines (VPNs), 88
virtualization, 81
visibility of clients, 1046-1047
Vista objects, namespaces, 631
VMAs (virtual memory areas), 343
VMs (virtual machines), operational vulnerabilities, preventing, 79
VMS (Virtual Memory System), 626
VPNs (virtual private networks), 88
vreply() function, 424
vsnprintf() function, 424
Vulnerability in Filtering a Character Sequence listing (8-25), 437
Vulnerability in Filtering a Character Sequence #2 listing (8-26), 438
Vulnerable Hex-Decoding Routine for URIs listing (8-5), 404
vulnerabilities
accountability, 40-41
authentication, 36
insufficient validation, 38
untrustworthy credentials, 37
authorization, 39
availability, 48-49
encryption, 43-45
integrity, 47-48
operational vulnerabilities, 76
access control, 69-70
attack surfaces, 68
authentication, 75
default site installations, 75
development protective measures, 76-79
directory indexing, 74
exposure, 68-73
file handlers, 74
host-based measures, 79-83
HTTP request methods, 73
insecure defaults, 69
network profiles, 73
network-based measures, 83-89
overly verbose error messages, 75
public-facing administrative interfaces, 76
secure channels, 71-72
spoofing, 72
unnecessary services, 70-71
Web-specific vulnerabilities, 73-76
operational vulnerabilities, 67-68
operators
right shift, 272-275, 277
size, 271-272
pointers, 280-282
software, 4-5, 18
bugs, 4-5
classifying, 14-17
data flow, 18-19
design vulnerabilities, 14-15
environmental attacks, 21-22
exceptional conditions, 22
implementation vulnerabilities, 15-16
input, 18-19
interfaces, 21
operational vulnerabilities, 16
security policies, 5-7
trust relationships, 19-20
type conversions, 246-248
C programming language, 246-270
sign extensions, 248-265
vulnerability classes, 14-16

W
wait functions, 765
waitable timer, Windows NT, 769-770
Wang, Xiaoyun, 48
Warning header field (HTTP), 1020
waterfall models, 13
wcsncpy() function, 453
Web 2.0, 1083
Web applications
access control, 1057-1058
ASP (Active Server Pages), 1113
configuration settings, 1118
cross-site scripting, 1118
file access, 1115
file inclusion, 1116-1117
inline evaluation, 1117-1118
shell invocation, 1115
SQL injection queries, 1113, 1115
ASP.NET, 1118
configuration settings, 1121-1123
cross-site scripting, 1121
file access, 1119-1120
file inclusion, 1120
inline evaluation, 1121
shell invocation, 1120
SQL injection queries, 1118-1119
auditing, 1078-1081
activities to isolate, 1079
avoiding assumptions, 1080
black box testing, 1079
enumerating functionality, 1081
goals, 1081

1171
multiple approaches, 1080
reverse-engineering, 1081
testing and experimentation, 1080-1081
authentication, 1056-1057
authorization, 1057-1058
business logic, 1041
C/C++ problems, 1075
CGI (Common Gateway Interface), 1009-1010, 1086
environment variables, 1087-1093
indexed queries, 1086-1087
client control, 1047-1048
client visibility, 1046-1047
dynamic content, 1009
cryptography, 1058-1059
HTML (Hypertext Markup Language), 1009
HTTP (Hypertext Transport Protocol), 1009
authentication, 1033-1036, 1056-1057
cookies, 1036-1038
embedded path information, 1022-1023
GET method, 1023, 1026
parameter encoding, 1026
POST method, 1025-1026
query strings, 1023-1024
Perl, 1093
cross-site scripting, 1096
file access, 1094
file inclusion, 1095
inline evaluation, 1095-1096
shell invocation, 1095
SQL injection queries, 1093-1094
taint mode, 1096
phishing and impersonation, 1059-1060
PHP (PHP Hypertext Preprocessor), 1096-1097
configuration settings, 1104-1105
cross-site scripting, 1103
file access, 1098-1099
file inclusion, 1101
inline evaluation, 1101-1103
shell invocation, 1099, 1101
SQL injection queries, 1097-1098
presentation logic, 1040-1041
redundancy, 1040
security environment, 1075-1078
server-side scripting, 1013-1014
sessions, 1049-1052
security vulnerabilities, 1051-1052
session management, 1052-1053
session tokens, 1053-1056
SQL injection, 1061-1062
parameterized queries, 1062-1063
prepared statements, 1062
second order injection, 1064-1065
stored procedures, 1063-1064
testing for, 1065-1066
SSIs (server-side includes), 1011
static content, 1009
Struts framework, 1008
threading issues, 1074
URLs (Uniform Resource Identifiers), 1009
Web server APIs, 1010-1011
XML injection, 1069-1070
XPath injection, 1070-1071
XSLT (Extensible Stylesheet Language Transformation), 1012
XSS (cross-site scripting), 1071-1074
Web Distributed Authoring and Versioning (WebDAV) methods, 1022
Web server APIs, Java servlets versus, 1106
Web servers
APIs, 1010-1011
directory indexing, 74
server-side scripting, 1013-1014
server-side transformation, 1012
SSIs (server-side includes), 1011
Web Services, 1084
  AJAX (Asynchronous JavaScript and XML), 1085
  REST (Representational State Transfer), 1085
  SOAP (Simple Object Access Protocol), 1085
Web Services Description Language (WSDL), 1084
Web tier (Web applications), 1042, 1044-1045
Web-specific vulnerabilities, applications, 73-76
  authentication, 75
  default site installations, 75
  directory indexing, 74
  file handlers, 74
  HTTP request methods, 73
  overly verbose error messages, 75
  public-facing administrative interfaces, 76
web.config file, ASP.NET, 1121-1123
WebDAV (Web Distributed Authoring and Versioning) methods, 1022
Weil, Alejandro David, 885
WEP (Wired Equivalent Privacy), 84
white-list filters, metacharacters, 435-436
Whitehead, Alfred North, 67
Wi-Fi Protected Access (WPA), 85
WideCharToMultiByte( ) function, 451-452, 457
width, integer types, 206, 226-227
Wilson, Daniel H., xvii
window scale option, TCP (Transmission Control Protocol) processing, 889
window station, IPC (interprocess communications), 690
Windows functions, Unicode, 450-457
Windows Internals, 4th Edition, 628
Windows messaging, IPC (interprocess communications), 689-698
  DDE (Dynamic Data Exchange), 697
  desktop object, 690-691
  shatter attacks, 694-697
  window station, 690
WTS (Windows Terminal Services), 697-698
Windows NT, 625, 627
COM (Component Object Model)
  Active X security, 749-754
  application IDs, 728
  automation objects, 729, 749
  CLSID mapping, 728
  components, 725-727
  DCOM Configuration utility, 732
  interfaces, 727-728
  OLE (Object Linking and Embedding), 728
  proxies, 730
  stubs, 731
  threading, 729-730
  type libraries, 731
DCOM (Distributed Component Object Model)
  access controls, 734-736
  application audits, 741-749
  application identity, 732-733
  application registration, 741-743
  ATL (Active Template Library), 740
  DCOM Configuration utility, 731-732
  impersonation, 736-737
  interface audits, 743-749
  MIDL (Microsoft Interface Definition Language), 738-740
  subsystem access permissions, 733-734
development of, 626
event objects, 767
file access, 659
  canonicalization, 663-666
  case sensitivity, 673
  device files, 666-668
  DOS 8.3 filenames, 673-674
  extraneous filename characters, 670-672
  File I/O API, 661-667
  file open audits, 674-675
  file squatting, 662-663
  file streams, 668-670
  file types, 668
  links, 676-680
  permissions, 659-661
IPC (interprocess communications), 685
COM (Component Object Model), 725-754
DDE (Dynamic Data Exchange), 697
desktop object, 690-691
impersonation, 688-689
mailslots, 705-706
messaging, 689-698
pipes, 698-705
redirector, 686-688
RPCs (Remote Procedure Calls), 706-724
security, 686-689
shatter attacks, 694-697
window station, 690
WTS (Windows Terminal Services), 697-698
KOM (Kernel Object Manager), 627
multithreaded programs, synchronicity, 813-815
mutex objects, 706
namespaces, 629
objects, 627-629
  boundary descriptor objects, 631
  handles, 632-636
  namespaces, 629-632
  nonsecurable objects, 629
  SymbolicLink objects, 629
  system objects, 628
origins of, 626
INDEX

pipes
  anonymous pipes, 698
  creating, 699-700
  impersonation, 700-703
  named pipes, 698-699
  permissions, 698-699
  pipe squatting, 703-705
POSIX subsystem, signals, handling, 784
  processes, 654
    DLL loading, 656-658
    loading, 654-655
    process synchronization, 765-770
    services, 658-659
    ShellExecute() function, 655
    ShellExecuteEx() function, 655
registry, 680
  key permissions, 681-682
  key squatting, 682-684
  predefined keys, 681
  value squatting, 682-684
RPCs (Remote Procedure Calls)
  ACFs (application configuration files), 710
  application audits, 722-724
  connections, 706
  context handles, 718-721
  DCE (Distributed Computing Environment)
    RPCs, 706
  IDL file structure, 708-710
  impersonation, 716-717
  MIDL (Microsoft Interface Definition Language), 708
  ONC (Open Network Computing) RPCs, 706
  proprietary state mechanisms, 721
  RPC servers, 711-716
  threading, 721
  transports, 707-708
security descriptors, 647-648
  access masks, 648-649
  ACL inheritance, 649
  ACL permissions, 652-653
  programming interfaces, 649-652
  strings, 651-652
semaphores, 768
  sessions, 636-647
    access tokens, 639-645, 647
    logon rights, 638
    SIDs (security IDs), 637-638
  threads, 654
  waitable timer, 769-770
Windows registry, path metacharacters, 420-422
Windows System Programming, 654
WinObj, 629-630
Wired Equivalent Privacy (WEP), 84
Wojtczuk, Rafal, 577
working directories, UNIX, 574
working papers, application review, 103-104
WPA (Wi-Fi Protected Access), 85
writing to files, stdio file system, 555-556
WSDL (Web Services Description Language), 1084
  _wscanfw() function, 391
WTS (Windows Terminal Services), Windows messaging, 697-698
WWW-Authenticate header field (HTTP), 1020
WWW-Link header field (HTTP), 1020
WWW-Title header field (HTTP), 1020

X

XER (XML Encoding Rules), ASN.1 (Abstract Syntax Notation), 983-984
XFS6_SVGA servers, privileges, misuse of, 478
_xlate_ascii_write() function, 354
XML (eXtensible Markup Language)
  encoding, 443-444
  injection, 1069-1070
  XML injection, 1069-1070
XPath injection, 1070-1071
XPath injection, 1070-1071
XSLT (Extensible Stylesheet Language Transformation), 65, 1012
XSS (cross-site scripting), 1071-1074

Y–Z

Yu, Hongbo, 48
Zalewski, Michael, 256, 546, 577, 806, 877, 880
zero extensions, 226
Zero-Extension Example listing (6-15), 258
zero-length fragment, 909
zones, DNS (Domain Name System), 986-987