

Developing Secure Internet Applications

CHAPTER

12

IN THIS CHAPTER

- **Common Sources of Programming Mistakes** 375
- **Metacharacters** 376
- **Exploiting Executable Code** 381
- **Application-Level Security** 387
- **Coding Standards and Code Reviews** 394

A great deal of this book has dealt with how to implement security using different flavors of standard components, whether they are operating systems, network devices, or protocols. This chapter is a bit different because it deals with custom components—specifically, Web applications.

Most organizations that are serious about electronic commerce or other Web applications tend to write some or part of the applications themselves, or have them developed by third parties. For example, your average Internet banking application isn't something that has been purchased off the shelf and loaded onto a couple of servers. It actually has been custom-designed and coded, most often by a group of people (developers are people, after all) whose main focus is not, unfortunately, security.

You can't really fault developers for not making security a priority. They're typically driven by deadlines and functionality, with launch dates for their Web products set by folks in marketing or sales or management. What's important from our perspective is not the developers' motivations, but rather, the resulting vulnerabilities that they can inadvertently introduce into their code and, therefore, the overall applications. These vulnerabilities can create an opportunity for an attacker to circumvent the security of the overall Web application without ever touching an operating system, spoofing a router, or hopping past a firewall.

Coders can make mistakes that can introduce vulnerabilities in many ways. For example, suppose that an electronic banking customer logs onto a banking application. To keep track of the customer, the coder embeds the bank account number into a cookie stored on the client side. When the customer checks his balance, the coder reads the value from the cookie to get the customer's account number so that it can query the database. The customer, guessing how this works, changes the account number and, whoosh, is now able to access a different customer's account information. This actually used to be a common mistake made by developers, and it illustrates how a system that is extremely secure from the standpoint of policy, operating systems, firewalls, routers, and other infrastructure can be rendered functionally insecure if application security isn't taken into account.

Internet application development is often the place where major insecurities are introduced. Many headline-grabbing security incidents have been the direct consequence of poor application security. This chapter discusses best practices for developing Internet applications. Although many secure programming topics are necessarily operating system-specific, and although a thorough list of these topics far exceeds a single volume (let alone a single chapter), we plan to arm you with enough knowledge to know *how* the general topics of programming errors impact security. By doing this, we hope to bring together the most common coding and development mistakes and present the best-practice solutions to these mistakes.

Common Sources of Programming Mistakes

Software developers make mistakes in several broad categories when developing Internet applications. As a general rule, almost all of these can be boiled down to not exercising enough care when handling user-provided data. The simple example of an account number being stored in a cookie on the client side is this type of error. Poor handling of user-provided data can happen at all layers of the protocol stack, not just at the application layer. An analogous example of poor data handling at the protocol layer is the land attack, in which an attacker (user) crafts TCP packets with their IP addresses and ports set to identical values, thereby causing systems to crash or slow down. This problem has affected products from several major vendors, including Microsoft, Cisco Systems, and Sun Microsystems, and it was a problem only because the applications (really, TCP/IP stacks) that accepted this user data (TCP packets) didn't handle it appropriately (basically and infinite loop occurred).

The same type of problem can occur at the application layer. For this reason, user-provided data always must be treated suspiciously. This is not a philosophy that many software developers are taught, although the level of awareness among developers has increased over the last several years. This is probably because of the number of well-publicized events that involved manipulations of user supplied data, such as users changing electronic shopping card information and buying appliances and the like for a few cents, or transferring funds from bank accounts that weren't their own. This chapter details several of the most important best practices that can go a long way toward securing an application exposed to the Internet. Although the details of specific implementations might vary between different environments and programming languages, the core concept remains the same: Any software module or system that handles user-provided data must be written with extreme care to ensure that the user has followed the expected rules of the game.

In a sense, Web applications are unique because part of the application is not really under the coder's control. For example, if the Internet application employs HTTP and Web browsers for the client interface, something that is most often the case, the developer's ability to affect client-side security is severely limited because the client environment is largely out of the developer's control. In this case, all user data must be checked on the server side to ensure that the client has not fooled with anything. On the other hand, if a custom interface is provided to the client so that it can access the application, it is equally important to ensure that this application does not adversely affect the security of the client's system. In the former case, the developer is concerned with only how the security of the client environment affects the server environment; in the latter case, the developer must ensure that he does not adversely affect the security of the client environment.

Overall, this chapter discusses six major topics that form the basis of secure Web application development:

- Metacharacter usage
- Buffer overrun conditions
- String formatting
- Session management
- Credential checking
- Client-side data cleansing

In addition, we'll discuss the importance of coding standards and code review in the development process. By understanding the security implications of each of these topics and how to code their applications appropriately, the Web application developer will have a solid basis for creating safe and secure code.

Metacharacters

It is difficult to assign the title of Most Prevalent Security-Related Bug to any one type of coding shortcoming, but a strong challenger for the title would be poor handling of metacharacters. First, some definitions are in order. *Metacharacters* are characters that have special meaning to certain applications. For instance, in the generic UNIX Bourne shell, the semicolon character (;) can be used to separate multiple commands. In standard SQL, the asterisk character (*) acts as a wildcard to match anything, just as it does in DOS commands such as `find *.doc`. These metacharacters are a problem for developers because they can allow users to manipulate applications in ways that subvert the security of the system.

Danger of Metacharacters

The way in which metacharacters can cause problems can be illustrated by considering the single-quote character ('). In SQL, the single quote is used to delimit strings in queries and thus acts as a metacharacter. However, the single quote is also entirely legal in a lot of user-supplied data, notably names. For instance, let's say that a Web application prompts a user for his name, perhaps as part of a registration procedure. This name then is put into a SQL database by the application. Remembering that the single quote is a delimiter used by SQL, let's look at what happens if the user's name is Jane O'Smith. If the application tries to put this name directly into the database, the SQL statement will look something like this:

```
UPDATE Customers SET Name = 'Jane O'Smith'
```

This statement won't be interpreted in the way that we would like (adding Jane O'Smith into the database) because SQL will think that the data is "Jane O" because the ' character is a

delimiter (metacharacter). Worse yet, there's a trailing `Smith'` that throws a monkey wrench into the database and might very well crash the application with it.

Because the characters that can act as special metacharacters vary widely based on the application that is evaluating them, there is significant overlap between those characters that constitute “normal” characters and those that constitute metacharacters.

The special nature of metacharacters doesn't really seem like a security concern yet, but consider what someone clever might do using metacharacters. Suppose that, instead of just inputting their name as Jane O' Smith, a sneaky user decides to enter a complete set of database commands when prompted for his username. This user enters his name as Jane O' UPDATE Admins SET Name= 'gubb', Privilege = 'Administrator'. The application, expecting that what the input supplied is the user's name, creates the following string, which is sent to the back-end database server as a command, thereby turning the previous innocent SQL statement into this:

```
UPDATE Customers SET Name = 'Jane O' UPDATE Admins SET Name = 'gubb', Privilege
↳= 'Administrator'
```

The result of this statement is to add a new account called gubb with administrator privileges. Now you can begin to see the danger of metacharacters. In Microsoft JetSQL, this statement is completely legitimate; it's a technique called “batching” queries, in which a program is allowed to submit multiple statements as in single database request. The Jet engine will happily execute all valid SQL statements within this request. In effect, the unwary application programmer has given a sneaky user a way to submit commands directly to the back-end database!

Working Safely with Metacharacters

One solution to the previously illustrated metacharacter problem is to write an application that is aware of the way Jet operates and to screen all the input for any attempts at using Jet-specific metacharacters in the username. This is fine, but what about the myriad of other types of databases or back-end services that the programmer will come up against? Because of the myriad of applications and the fact that each has the opportunity to define its own metacharacters, it's largely futile to try to envision every possible metacharacter attack scenario and screen for it in the code. For this reason, we recommend a more generic approach to tackling the metacharacter problem. To protect an application from attacks that utilize metacharacters, the developer must be aware of three issues:

- What does “normal” input for each user-supplied value look like?
- For each user-supplied value, what about the data supplied that is considered “significant”?
- Which characters do back-end applications treat as “special”?

When approached in this order, a developer's job of adequately handling these funny characters is eased dramatically. When the developer understands these three issues, metacharacters can be addressed by removing their ability to cause special actions. This is accomplished by "escaping" the metacharacters.

Identifying Normal Input

First, the developer asks, what does *normal* input look like? Consider a phone number. For all intents and purposes, a phone number consists of some numbers and possibly a few punctuation characters. Depending on where in the world you are, phone numbers can take the form of (123)456-7890, 123.456.7890, 01-23-45-6-78, or maybe 12 3 456 78. Given these four formats, you can dramatically restrict the possible characters that you allow. So, instead of 127 possible characters (the base ASCII set), you have 16 characters (digits, the parenthesis, the period, the dash, and the space). Anything outside of this set can be rejected as invalid. This allows a developer to write a script to "clean" the input of any characters that fall outside of this range, or to simply reject any input that includes characters other than these 16.

Determining Character Significance

Second, the developer should consider what is *significant* about these characters. The punctuation is just there because it's easier to read for humans. In fact, when you dial a number, you never use these punctuation marks or spaces. When you're storing and handling this data inside of a program, the punctuation just gets in the way. Therefore, when a user submits the phone number, you can strip out these unnecessary characters and put them back in when presenting the data. Thus, the only significant characters are the digits, 0123456789. So, you've reduced your allowed character set to only 10. Now when you clean the input data, you can get rid of everything except these 10 digits.

Identifying Special Characters

Finally, you must think about which, if any, of these characters the back-end application considers *special*. In the case of the phone number, you would not expect any of this set of 10 digits to be a metacharacter. However, you saw that in the case of someone's name, this is not the case (the single quote was indeed a special character). In some cases, you might determine that some of the allowed characters in the input can be considered metacharacters by the application. For example, \$50.01 requires the characters \$ and ., which could very well be considered metacharacters by the back-end application. If you have to leave in characters that might be considered metacharacters, there are two main ways to deal with them: Escape them or delete them. Deletion is the most straightforward method; if a character doesn't belong or is potentially hazardous, just get rid of it. This is what we did in the first two steps described previously. Unfortunately, this not always can be done if the character is necessary, for example, if a license key for a software product can include \$, #, @, &, or other characters. In this case, you must escape the characters.

Escaping Metacharacters

Escaping metacharacters means that you strip them of their special meaning. This is done by placing an “escape character” before the metacharacter. This is where things get a bit tough to explain. Basically, you choose a character that will represent your escape. This character is placed before any potential metacharacter to say, “Treat this next character, which looks like a metacharacter, as just a plain old normal character.” An example is warranted here.

Some UNIX programs, such as `grep`, make use of the asterisk (*) inside the program itself. Now, like SQL, most UNIX shells treat the asterisk as a wildcard. So, for example, you can issue the UNIX command `grep c* myfile` to look for every word that starts with a `c` in the file `myfile`. The developers who wrote `grep` knew that the asterisk would be used as an input to the command, so they supplied mechanisms for users to say, “Hey, I know that this next character is special, but just treat it as normal.”

For example, how do you `grep` for anything in `myfile` that starts with an asterisk? You can’t say, `grep * myfile` because `*` is a metacharacter and that command would return everything in `myfile`. You need a way to tell the program that the next character isn’t a metacharacter. Aha! This is where the escape comes in. The most common of these escape mechanisms is the backslash (`\`). Whenever an application encounters a backslash, it treats the character that follows as a literal (normal character), not as special character. Astute readers probably are noticing that this `\` character is just another metacharacter. You’re right, it is. And to specify the “real” backslash, you precede a backslash with a backslash. The backslash can be used in SQL just the same as in the Bourne shell—that is, SQL recognizes that the `\` character is a metacharacter. What must you do with it? Well, the former application that accepted any data that the user gave must now do some data cleaning first. You still will be handling the data supplied by Ms. Jane O’Smith, but you will code in some steps that will escape the single quotes before submitting the data to the database. The first example that we gave (the “good” example) will look like this:

```
UPDATE Users SET Name = 'Jane O\'Smith'
```

This is entirely valid and will be accepted by the database without a problem. In this case, the developer has written code that looks for any metacharacters in the string (in this case, it finds the single quote) and adds an escape character before it. When this is passed to the database, the database “understands” that the data has been escaped and treats it as a normal character, not a metacharacter.

Now, let’s take a look at what the “evil” data will look like:

```
UPDATE Users SET Name = 'Jane O\' ' UPDATE Admins SET Name = \'gubb\', Privilege
=&#x2D; \'Administrator\''.
```

This command will add a very silly name into the database, but it won't execute any commands, which is what you want. Although it's silly, this value potentially could be handled incorrectly by another application down the line. So, just because hazardous data is handled properly on the way in, don't assume that everything is okay. The same vigilance must be part of *all* applications, whether they directly handle user input or whether they handle it on the way back, when it's coming from a trusted system.

In the world of metacharacters, the most common offenders that you, as a programmer, need to look for are these:

- ASCII `\xa` (newline)
- ASCII `\xd` (carriage return)
- % (percent)
- ; (semicolon)
- * (asterisk)
- ! (exclamation point)
- # (pound sign)
- \$ (dollar sign)
- & (ampersand)
- (and) (parentheses)
- { and } (curly braces)
- [and] (square brackets)
- " (double quotes)
- ' (single quote)
- | (pipe)
- ? (question mark)
- < and > (greater than, less than symbols)
- ` (backtick)

Again, this is not an all-inclusive list, but it is a good start.

Client-Side Data Cleansing

Note that many sites make use of client-side JavaScript to validate user input before it is sent to the server. This is a nice feature for the user because he won't have to wait for the server response to find out whether some of the submissions were incorrect or incomplete. For instance, if a user doesn't fill out a required phone number field, a JavaScript routine can catch this and alert the user before the form is submitted to the server. Input validation of this sort, however, is not sufficient when attempting to ensure that user input is safe from a security

standpoint. It is trivial to evade client-side JavaScript, either by simply disabling it in the browser or by submitting the request directly to the server. If data is not also stripped of invalid characters on the server, the potential for batched SQL queries, escape characters, and so on exists. All data cleansing should occur in the back-end application (either in addition to or in lieu of cleansing on the client side).

Exploiting Executable Code

Executable code exploits are in strong contention with metacharacters for the title of Most Prevalent Security-Related Bug out there. As with metacharacters, most developers have encountered executable code problems when debugging their code without realizing the security implications involved. Examples of this are the ever-popular Microsoft Blue Screen of Death or the more generic segmentation fault. These are both examples of what happens when a program changes data that is outside the bounds of the memory space that it has been assigned.

The C programming language is especially susceptible to this problem because it allows programmers to write to memory directly with pointers, without performing any bounds checking. Contrast this with other languages such as Java that perform more rigorous bounds checking to explicitly prevent this type of problem. Executable code exploits, such as buffer overflows, might allow an attacker to introduce and run his own executable code rather than simply issuing commands to pre-existing applications or system calls. This means that an attacker can upload his own applications and subvert the security of the existing application, all at the same time. In this section, we examine two of the most prevalent types of executable code exploits: buffer overruns and format string bugs.

Buffer Overruns

A buffer overrun occurs whenever an application allocates a specific amount of memory (a “buffer”) and then later copies more data into this buffer than there is room for. For instance, suppose that a programmer reserves 256 bytes in memory for the name of a file, thinking that no filename ever could be longer than 256 bytes. Then later it turns out that a file is 40 directories deep, and the name of each one of these directories is about 8 characters (not to mention the name of the file itself). So, when the program tries to copy the more than 320 bytes of the name into a buffer that was designed to hold only 256 bytes, something’s got to give. What normally occurs is that the data that directly follows the buffer in memory is overwritten. In this case, it turns out that computers are fairly stupid; they are capable of doing only what users tell them to and they don’t really think about the consequences. So, when a programmer says, “Take this stuff and copy it over that buffer,” the computer just does it. Whether “over there” is capable of holding whatever happens to be “over here” is left up to the programmer. And that’s where the problems start.

Buffer overruns manifest themselves often when dealing with strings in the C programming language. Most C libraries have a set of functions designed specifically to deal with strings, and they handle strings by treating them as an array of characters terminated by the NULL (ASCII value `0x0`) character. One of the most rudimentary of these functions is `strcpy()` (string copy, in case you couldn't guess), which copies a string from one location in memory to another.

`strcpy()` takes two parameters: the address of the location in memory where the strings are to be copied to and a pointer to where in memory the string should be copied from (the destination and the source locations). `strcpy()` does its job by copying the contents of memory at the source location into the destination location, doing this byte for byte. `strcpy()` keeps doing this until it finds a NULL character in the source string, which it takes to mean that the end of the string has been reached. If it turns out that the source string is longer than the amount of memory that has been reserved by the destination pointer, something is going to get overwritten. `strcpy()` is dumb and has no idea how big the buffers are. It just copies data. So, if the programmer does not make certain that enough room has been allocated to take the entire string being copied, a buffer overrun occurs. The result? Whatever happened to be in memory *next* to the allocated buffer gets overwritten. This is probably better explained by Figures 12.1, 12.2, and 12.3.



FIGURE 12.1

Destination buffer before copy.

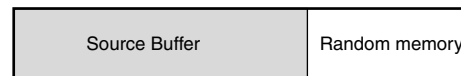


FIGURE 12.2

Source address before copy.

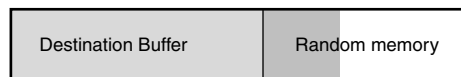


FIGURE 12.3

Destination address after copy.

When a buffer has been overwritten, all sorts of nasty things can happen. For instance, the capability to change the value of variables in memory can have serious repercussions. For

instance, what if a filename was stored next to the destination string buffer? If you, as an attacker, knew how the filenames and permissions were set up in memory, you could utilize a buffer overflow to change the name of that file and possibly its permissions to gain access to a protected file that you couldn't normally access.

Here's a simpler example: Let's say that the next value in memory is the bank account number that the application is going to transfer money to in conducting a transaction. If you can create an overflow condition that changes the bank account number from your own to, say, Bill Gates's, you can transfer funds from his bank account to pay for that new big-screen TV.

A wider class of buffer overflow attacks utilizes the fact that computer subroutines often store information about the next program line to execute in memory. In layman's terms, when a routine executes, it looks into memory for where to go next (what to execute next). If you can create a buffer overflow that overwrites this area of memory, called the stack, you can redirect the subroutine. The best type of buffer overflow happens when you not only put a redirection into memory, but also put new program code into memory using the same overflow. So, for example, you can create some code that gives you root privileges and then use a buffer overflow to place that code into memory and redirect the computer to execute that code.

An Example: String Functions in C

To illustrate how difficult it can be to protect against buffer overflows, let's look at a widely used class of functions and delve into some detail on how they can be used, both inadvertently and purposefully, to create buffer overflow conditions.

"Dumb" functions such as `strcpy()` aren't the only way to trigger a buffer overflow. *Any* code that copies data from one area of memory into another can overrun an allocated buffer. For instance, consider the following fragment of C code:

```
for(i = 0; i < length; i++)
    buf1[i] = buf2[i];
```

This code fragment copies `length` number of data from `buf1` into `buf2`. If the variable `buf1` isn't at least `length + 1` characters long (remember, you need the `NULL` terminator), you will get an overflow. The following is a list of commonly found C library functions that can cause overflows:

- `strcpy()`, `strncpy()`
- `sprintf()`, `snprintf()`
- `strcat()`, `strncat()`
- `memcpy()`, `memmove()`
- `fgets()`, `read()`, `fread()`

Note that some of the functions listed (`strncpy()`, `strncat()`, and `snprintf()`) specifically were developed to help protect against buffer overruns. Each of these functions is a modified version of its generic counterpart (`strcpy()`, `strcat()`, and `sprintf()`, respectively) that accept an additional parameter: the size of the destination buffer. However, it is *still* up to the programmer to provide the correct value for this parameter. Consider the following:

```
strncpy(dst, src, strlen(src));
```

The previous function uses the `strlen()` function to calculate the length of the source string (`src`), and this becomes an input to the `strncpy()` function. The problem here is that the `strncpy()` function just looks for a null terminator to figure out the length of the source string. This is a common technique used by programmers, but it will have the exact same effect as just calling `strcpy()` because it's still possible for the source string to be longer than the destination buffer! The correct way to call `strncpy()` is to specify the `length` parameter as the size allocated for the `dst` parameter, *minus 1*. Remember, you have to leave room for the NULL terminator at the end of the string.

Most implementations of `strncpy()` copy *exactly* the number of bytes specified by `length`. The result is that the destination will not include the NULL terminator, and `dst` will be what is called an unterminated string. If someone later uses the `strlen()` function to find the length of this string, `strlen()` will keep searching in memory until it finds a NULL character and so will return the wrong value. This, in itself, could result in a buffer overflow or cause data to be handled in an insecure way. To cause additional headache for programmers, the way that `strncpy()` copies data is not standard among all `strn` functions. In fact, there really is no standard behavior. For example, most implementations of `snprint()` *will* add a NULL character to the end of `dst`¹. Because it might not always be possible for a programmer to know exactly how a function will be implemented given any operating system, it is recommended that the programmers manually terminate all strings, just to be safe.

How Buffer Overflows Are Utilized by Hackers

This previous example illustrates how much care must be taken by software developers when writing code. You might ask, "How does this turn into a security problem?" The problem occurs whenever a function accepts user-provided data and that function has been written so that a buffer overflow can be created by a crafty user. This data can come from two basic sources. The first source is client data that is provided to an application. The second is remote procedure calls (RPC) in which one machine can execute procedures on a second machine by passing it data, having the remote system process the call and have the result returned to it.

A good example of the former case is the Microsoft ISAPI overflow that was utilized by the well-publicized Code Red attack that occurred in the summer of 2001. In this case, the vulnerable code was a dynamic link library (DLL) called `idq.dll` that resided in the Microsoft

Indexing Service; it was loaded by default with the Internet Information Server (IIS) Web server. This allowed anyone on the Internet to access the vulnerable code through a hand-crafted URL sent to the server.

The URL contains a large amount of bogus data (the XXXXs) that are used to create the overflow condition, along with a block of starter code that points to a much larger block of executable code. In this particular case, the goal of the code was to install a program on the target system that then would go out and scan for other vulnerable servers and attempt to infect them. Keep in mind that somewhere in the ISAPI DLL was a function written by a developer who assumed that the function always would be given “clean” data to process, a mind-set that all developers now should be getting over.

Just as a user can provide data to a function that can create overflow conditions, applications that employ RPC architectures can create the same condition by sending function arguments across a network to a target machine running a vulnerable application. For example, UNIX RPC has been a notorious source of many buffer overflows. A procedure such as `statd`, which provides system statistic, and `nfsd`, which is part of the Network File System, have allowed a generation of hackers to successfully “own” legions of UNIX systems.

Format String Bugs

Format string bugs have traits similar to both buffer overruns and incorrect metacharacter handling. Format string errors manifest themselves when a function that takes a variable number of arguments inadvertently is given *too few* arguments. To illustrate how format string bugs can occur, let’s look at another C language class of functions, the lib C `printf()` functions. As a reminder to all of you out there who aren’t up on your `printf` trivia, this function expects arguments in the following form:

```
printf(formatstr, var, ...)
```

For example, `printf("Hello %s", name)` would print “Hello Bob,” assuming that the variable `name = "Bob"` and the `%s` are formatting variables that say, “Insert a string here.”

Now consider the following fragment of C code:

```
void my_function(char *somestring)
{
    printf(somestring);
}
```

In this perfectly valid code, the programmer has assumed that `somestring` will be a literal value that needs to be printed. If `my_function()` is called as follows, everything is fine:

```
my_function("hello, world");
```

In this case, the string "hello, world" would simply be printed to standard output. Note that, in this example, no formatting string is used and none is assumed to be present in the argument passed to the function. However, if `my_function()` is called in the following manner, problems creep in:

```
my_function("hello, %s");
```

When `printf()` is invoked by `my_function`, it sees `printf("hello, %s")` and expects that a second argument to the function would be substituted for the `%s` formatting code (as in the "Hello Bob" example). However, there is no second argument, so what will `printf` do? It looks for this second parameter, which typically is stored on the stack. Because no second parameter was pushed onto the stack by the calling function (in this case, `my_function`), `printf` actually will pull some other data resident on the stack.

How can this type of bug be turned into a security hole? Well, imagine that an attacker has access to the source code or is able to reverse-engineer the executable code and so knows that this bug occurs. He might be able to learn that a filename, password, or encryption key is sitting in a local variable on the stack. Knowing this, he can cause this sensitive data to be printed using the `printf` function. This is, of course, a simple and contrived example, but it should be obvious that other, more complex examples could yield similar results. One of these complex examples can be found in a little-known format character specification in the `printf()` functions: `%n`. Let's take a look at how this can be exploited to create bigger and better buffer overflows.

The `%n` specification instructs the `printf()` function to place a number of characters into a variable in the function call. So, a call such as this would place the value 5 in the variable `someint`:

```
printf("hello%n", &someint);
```

Now let's combine this with the previous example (yes, we *are* trying to stretch the bounds of your resolve). Suppose that you call `my_function` in the following manner:

```
my_function("sdaffkd1fjhdfkjhsdkfhdfjsdfjkfk %n").
```

The `printf` function that is called by `my_function` gets this:

```
printf("sdaffkd1fjhdfkjhsdkfhdfjsdfjkfk %n").
```

It then counts the number of characters before the `%n`, which, in this case, is 32, and assumes that a pointer exists on the stack pointing to the location in memory where the 32 will be stored. In this case, there is no pointer, so `printf()` treats the next 4 bytes on the stack as the pointer and stores 32 at that location. So, this particular oddity has allowed the attacker to put an arbitrary value (32) into a location in memory. Without going any further on the technical front, let's suppose that the memory value that the attacker overwrote is a seed to a pseudorandom number generator. If the attacker knows the seed, he can predict what the pseudorandom

number generator will output (that's why it's called *pseudorandom*). If this is used to create a session for an HTTP session, the attacker might be able to hijack a seemingly secure Web application.

A Final Word on Executable Code Exploits

It should be obvious that attacks such as buffer overruns and format bugs require an in-depth understanding of programming languages, memory utilization, and program execution on the specific platforms that the attack is designed to exploit. Typically, the attacker has access to either source code (for open-source systems) or the tools required to reverse-engineer executable code (for systems whose source code is not publicly available). Given the level of sophistication required to create such an exploit, it would seem odd that so many executable code exploits have been used by attackers; however, when an exploit is written, it requires only a moderate level of expertise to utilize the exploit against a vulnerable system. The Code Red exploit is an example of how the utilization of a well-documented buffer overrun was automated so that no expertise was required for it to propagate to new systems.

Application-Level Security

Thus far, we have concentrated our discussion on ways in which unsafe coding practices can have ramifications on other applications or operating systems. However, additional considerations must be made to protect the application from itself. This is particularly true in browser-based applications that function over the Internet, usually making use of HTTP as the communication protocol.

Although HTTP generally runs on top of the session-oriented TCP, it is woefully void of any concept of state. By this, we mean that each HTTP command is not dependant upon the previous command, and the server generally does not keep track of the order in which commands are presented to it. Because the HTTP connections are not persistent and each request takes place independently (from a connection standpoint), application designers are presented with a special problem of somehow correctly correlating these requests and defining some sort of an idea of "user session." This is much different than, say, a TCP connection, which is marked by a start or *SYN* and end or *FIN*. It would be nice if, when a user starts to browse to a Web site, he could signal, "Hey, I'm user XYZ and I'm starting to browse," and then could signal that he was leaving the site when he moved on. This is not, however, the way HTTP works. Every request is a start and finish unto itself.

Cookies

Cookies were created to aid in this effort and to create a rudimentary implementation of state at the application layer. To review, when a user makes an HTTP request, the server returns the appropriate response as well as a unique value, a cookie, that positively identifies the user.

Subsequent requests from the user contain this unique value that is read from the cookie stored by the user's browser, allowing the server to keep track of the session. Cookies make it easy for developers to keep track of state because most of the work is done by the Web server software itself. However, cookies are insufficient from a security standpoint. That is, using cookies on their own as an authentication mechanism is terribly ineffective (and was explicitly *not* their intended use when originally developed).

Cookies are often deterministic, meaning that they are constructed using very simple rules that can be guessed by an adversary. For example, suppose that you're at a hotel and you log on to your banking site using your username and password. You then get a cookie that is formed by concatenating your first name with your account number. Now, in the room next to you is a wily hacker (let's say that you're in Vegas and the Defcon conference is in town) who decides that he's going to steal your session. Because your first name and account number are easily obtained, he forges a cookie and then browses to your bank site. The bank's server sees the cookie and assumes that the hacker is you—voilà, your session has been hijacked.

This attack is simplified because cookies persist through user sessions and are stored in the clear on the client's system. They also have no challenge mechanisms built into them and are easily forged by an adversary. So, let's say that you log into your bank site from a kiosk in the lobby of the hotel (now you really deserve what you get). When you're finished, your cookie is most likely still on the computer! The next user might easily utilize the cookie, either to try to hijack your session or to study how the particular site creates cookies to forge an attack at a later time. Cookies can be used to track user sessions without any negative security impact, but they most certainly cannot be relied upon for authentication.

Source IP Addresses

You might be tempted to make use of the source IP address to correlate user sessions. This seems like a good idea on the surface because the source IP would seem to be something that identifies a specific user's machine, at least for the duration of the session. However, this is ineffective for multiple reasons. First and foremost, using a source IP won't work if you have multiple clients coming from the same source IP address, as is the case when clients are behind a firewall using hide-mode network address translation or a proxy. Along these lines, many hide-mode address-translation implementations operate from a pool of source IP addresses, so a client might have a different address every time he makes a request.

Additionally, by making use of a proxy, it is trivial for a user to pretend to be someone else. Therefore, no stock can be placed in IP addresses as a form of identification. It can be worthwhile to keep track of client IP addresses for audit purposes when developing applications that run on top of HTTP; however, you should never use them for identification or authentication purposes.

Effective Session Management

The best way to enable session authentication when hampered by HTTP is to augment the cookie with something that is more dynamic and that can't easily be forged. We call this a multifaceted authentication token. The idea behind this type of token is to create a set of variables, at least one of which is constantly changing. For instance, when creating browser-based applications, each request coming from the client should contain at least two credentials: a *client ID* that uniquely identifies the client's account (such as username, account number, and so on) and a constantly changing *session ID*. Every time the client makes a request, the server checks the client ID and the session ID to verify that the session is valid and belongs to the client.

For example, let's say that you log back into your banking application. The server returns to you a client ID that is JOHNPUBLIC and a session ID that is 14589 (right now, let's not worry about how the session ID is created). The next time the browser makes a request to the server (such as when you start to perform a funds transfer), it sends both the client ID and the session ID. The server checks the session ID and sees that it's the same one that it just gave you, so it carries out the browser's request. When the server returns the information to your browser (such as your new account balance), it hands the browser a new, randomly generated session ID—say, 83245. The next time your browser makes a request to the server, it presents `ClientID = JOHNPUBLIC` and `SessionID=83245`. Thus, the session ID is constantly changing. Again, the server checks the session ID against the last one that it sent to you. If someone gets hold of an old session ID, it's useless because the server “knows” that it already has been used and, therefore, is expired. This type of mechanism can cut down on session hijacking because the session ID is constantly changing.

Now, in introducing the session ID, you have to be careful: If an attacker can predict the session ID, he can indeed steal the session from the valid user. For example, if each session ID is created by incrementing the previous session ID by 50, it's not going to be hard for an attacker to forge a session ID. In addition, it's going to be easy for an attacker who can steal a few session IDs to figure out the ID creation mechanism (let's see, you steal one session ID that is 18543, the next one you steal is 18593, and then you miss a few but catch one later that is 18643—hmmm, how are they being generated?).

A good way of creating a new session ID is by utilizing a random number-generation program and a one-way function such as a cryptographic hash function (such as MD5). Why, you might ask, don't we just use the operating system-supplied RAND function to generate a random number? Why use the hash? There are a few reasons. The first is paranoia: You can go from about 4 billion possible values for a session ID (using a 32-bit number produced by RAND) to 18,446,744,073,709,551,615 possible values (using a 64-bit hash output) with very little performance impact. Second, by using a function in the creation of the session ID, you have an

effective “stub” in case you decide to move to a challenge-response mechanism, such as some kind of digital certificate–based authentication scheme. Finally, it turns out that most random number generators that are provided as operating system calls aren’t so random after all; by utilizing the combination hash function, you can get an additional layer of protection.

Replay Attacks and Session Security

As previously stated, the session key value is constantly changed, normally after each request. This prevents replay attacks in which an attacker tries to use an old session of the client to talk to the server. Changing the session key in this way limits the use to which a compromised session key can be put. For instance, if an attacker is capable of intercepting a session key (by either sniffing the network or reading it from the client’s machine), that session key will be valid for one only client-to-server request.

In addition, the session ID can be used only if the attacker reads it as it comes from the server (if the key was intercepted when sent by the client, it would become useless to the attacker because the server “nullifies” it after processing the request). Changing the session key with every request limits the amount of damage that can be done by an attacker to one request only (because the next session ID will be different). In addition, this scheme is a fail-closed mechanism. For example, in the scenario in which a key successfully is compromised and the attacker successfully issues a request to the server, the real client’s session will be desynchronized. The next time the valid user sends a request, he will find that this session ID has expired. This is easier to understand by looking at Figure 12.4.

When this desynchronization takes place, two things happen. First, the client is forced to establish a new session. Although this will be an annoyance for the user, it also will shut down the attacker’s ability to make use of the compromised session. Second, the server will be aware that the session became desynchronized and can record this for audit purposes or can take additional steps. For instance, if the server application notices that the sessions from a specific client become desynchronized at a higher rate than normal, this could indicate that there is something wrong on the client side or that perhaps something more insidious is occurring.

Credential Checks Within the Application

Although being able to verify that a request is generated by a valid client with a valid session is important, this by no means is all there is to Web application security. Within the application, attempted client actions must be compared to access control lists that define what actions the clients are allowed to take. Take, for example, help-desk software. Usually clear roles are defined within the context of the application, the role of a manager, the role of an administrator, the role of a help-desk operator, and so on. Each of these roles will need to perform different actions and, more importantly, should be restricted from performing other actions. For

instance, the operator should require access to only the trouble tickets currently open and assigned to him. A manager, however, would need access to all open tickets.

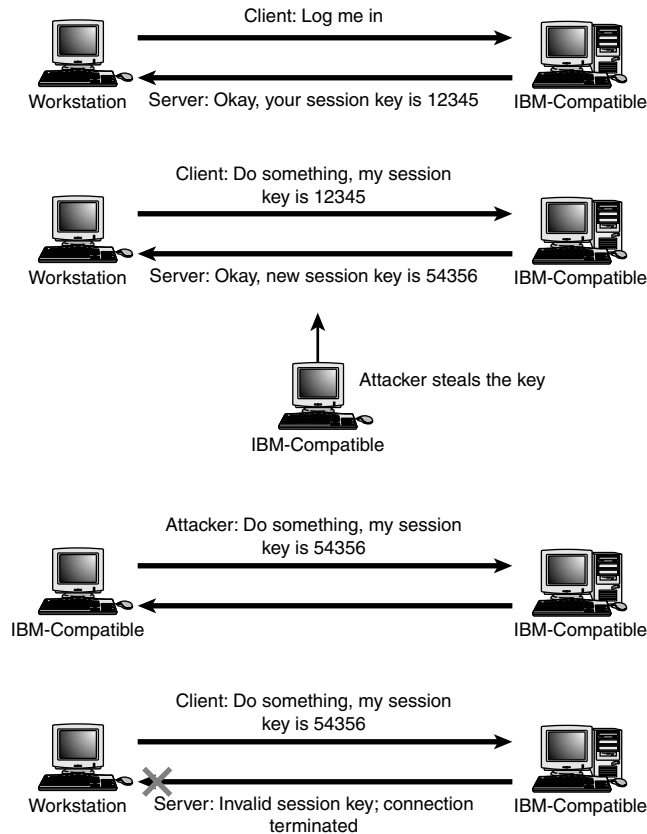


FIGURE 12.4

Desynchronization of client session.

When developing role-based security of this type, a great deal of planning must take place to properly define roles, assets, and the interaction between the two. This might seem obvious, but without sufficient planning and also room for further expansion, a credential system easily could turn into something insufficient and unusable.

Example: Access Control for a Trouble-Ticketing System

As an example, let's look at a problem that confronts many Web application developers: user access control. The intent of this example is to show that a Web application developer must

plan ahead during the design process to determine *exactly* what a user will be able to do. The decisions that the developer makes at this stage of the application design will carry through the entire coding phase and have a significant effect on the overall security of the final software system.

One way of creating effective access controls within an application is to utilize a model based on a slimmed-down version of the Windows Discretionary Access Control List methodology. In this paradigm, each user object has a list of credentials associated with it, and these credentials either can be explicitly granted (user-level credentials) or can be inherited based on the type of user (role-based security). In addition, each object or asset has a list of credentials (usually read, modify, delete, and execute) associated with it. Whenever a user object attempts to perform some action on some asset object, the user's credentials are checked to see whether the action should be allowed. A paradigm such as this is sufficient for moderately complex systems, is easy to implement, and is quite extensible.

Let's take a look at a rudimentary implementation of this. Using the help-desk software example from before and our three roles (operator, manager, and administrator), let's apply the credential-checking methodology for three assets:

- Trouble tickets
- User accounts
- Client records

First, you need to define the structure used by all assets. This consists of a description of the asset, an ID that specifies the owner of the asset, a unique identifier (ID) for the asset, and so on.

```
/* structure to define an asset */
struct asset{
    u_char description[MAXDESCSIZ + 1]; /* internal description of asset */
    u_int owner; /* ID of object owner */
    u_int asset_id; /* unique ID of object */
    ... /* sundry other asset info */
};
```

Next, define the permissions that can be applied to the asset:

```
/*
 * type of permissions. we implement the permissions as a list of
 * explicit permissions or'd together as an unsigned integer
 */
#define PERM_READ 0x00000001
#define PERM_CHANGE 0x00000010
```

```

#define PERM_DELETE      0x00000100
#define PERM_EXECUTE    0x00001000
#define PERM_CREATE     0x00010000

/*
 * structure to define an asset
 */
struct privilege{
    u_char description[MAXDESCSIZE + 1];    /* description of privilege */
    u_int  privid;                          /* unique privilege id */
    u_int  asset_id;                        /* ID of asset this privilege applies to */
    u_int  priv_types;                      /* types of privilege, e.g. read, write,
etc. */
    ...                                     /* sundry other privilege info */
};

/*
 * linked list of privileges
 */
struct privlist{
    struct privilege *priv;
    struct privlist *next;
};

#define ROLE_OPERATOR    1
#define ROLE_MANAGER    2
#define ROLE_ADMIN      3

/*
 * structure to define a user
 */
struct user{
    u_char description[MAXDESCSIZ + 1];    /* description of user */
    u_int  userid;                        /* unique user id */
    struct privlist *privs;                /* all privileges assigned to a user */
    u_int  role;                          /* type of user, e.g. operator, admin, etc.
*/
    ...                                     /* sundry other user info */
};

```

Although this code is not exactly ready for cut-and-paste use, it gives the general architecture for how to implement this type of system quite easily. Each user has a list of privileges. Each of these privileges links to a specific asset object and enumerates a list of permissions that the

user has. Each object simply has an ID and an owner. With just these small structures, you can offer a relatively granular privilege system that can control the following:

- Whether the user is allowed to read, modify, delete, or execute an that object he owns
- Whether the user is allowed to create new objects of a specific type
- Whether the user is allowed to read, modify, delete, or execute an object owned by someone else

You also can define assets and users of nearly any type. To put a finer point on it, you can create privilege objects for more abstract assets, such as, “Can this user access objects that it does not own?” and “During what hours may this user be logged in?”

Again, the point is to create a system that offers flexibility and extensibility both in granularity of permissions and in the definition of its various objects: roles, assets, privileges, and users. The system presented here is by no means complete, and it is not intended to present the “best” solution; it is offered mainly as an example of the various constituents of what an application requires for internal credential checking.

Coding Standards and Code Reviews

Many times in this book we have stressed the importance of policy, and with application development this is no different. Besides heeding all of the technical matters discussed in this chapter, it’s important that application-development teams have a baseline understanding of what’s expected from when it comes to writing code. For example, do you expect coders to write their functions so that they check the length and number of the parameters passed to them? Do you expect coders to include a default condition in every IF ... THEN or CASE statement so that if any of the expected conditions aren’t encountered, the code does not “fall through” into some unanticipated state?

Questions like these are best addressed in a coding standards document. Many good examples of such documents are freely available, making it easy to mix and match to come up with a document that fits your development environment. Some of the topics that should be addressed are listed here:

- What programming and scripting should be used
- How error messages should be formatted and how errors should be handled
- Function calling and return format
- Bounds checking
- Memory utilization
- Format and readability of code

- Function and variable naming
- Documentation (always a sore spot for developers)
- Configuration management
- How and what to test

We have found that most developers are eager to build the best code they can, but they are sometimes ignorant of simple practices, such as metacharacter cleaning, that can help them develop better and more secure applications. By creating a coding standards document and keeping it brief, you not only create a framework for security, but you also tip off developers to practices that they might never have guessed are insecure.

Having a coding standard also sets up your team for an important step in developing secure code: code review. Code review has a bad rep, perhaps because coders don't like anyone criticizing their work or perhaps because it often is carried out by people who are more interested in how code is formatted than what it does. During a code review, the source code or an application is read by someone other than its developer to look for bugs or deviations from the coding standard. Like anything else, having a second set of eyes look at your work often uncovers problems that you, as its creator, are blind to. In our experience, code review often has uncovered bugs such as insufficient bounds checking, unanticipated or unchecked error conditions, or opportunities for the introduction of buffer overflow conditions that otherwise would have gone unnoticed.

Code review can be done by having developers “swap code” with their peers (a peer review, if you want the formal term) or by having a third party conduct the review. In all cases, the development team needs to be a part of the review process. In addition, the overall application architecture should be part of the code review process and should be available during that process, to allow the reviewers to see how pieces of the application fit into the overall system architecture.

By developing coding standards in implementing code reviews, development teams can help ensure that their members follow good coding practices and don't inadvertently (or even purposefully) introduce security flaws into otherwise well-performing applications.

Summary

Security must be a consideration in the development of *all* software. What might not be considered critical to security now someday could be expanded or reused in a way that directly relates to security. Software bugs and shortcomings are arguably the source of *more* vulnerabilities than anything else. Many developers are not security engineers, and many security engineers are not developers. Thus, it is the responsibility of both parties to work together to educate themselves and educate each other. Security engineers need to become aware of more

than just the fact that buffer overruns exist and are bad, but also of the ways in which overflows manifest themselves in code and the proper way to guard against them. Likewise, programmers need to be very aware of the ramifications on the inner workings of what might appear to be perfectly benign function calls. How many programmers do you know who know what the return value from `sprintf()` is? How many more actually make use of it?

Good code comes from hard work, structured programming, extensive knowledge, and constant review. Development teams must design their coding standards not just from a readability and grammatical viewpoint, but also from a security viewpoint. Likewise, source code review and audits must be part of the development process not just to ferret out benign bugs, but also to check for security-related issues.

¹ Well, more precisely, it will insert a NULL character of length bytes from whatever `dst` points to.