CHAPTER 3

Secure Design Principles

While the previous chapter was concerned with high-level approaches and trade-offs in security, this chapter will focus on security design principles. When building a house, there are certain very specific things that a builder will do: roofing shingles are laid so that the higher shingles overlap the lower ones. Flashing is placed over the top of newly installed windows. These specific practices protect the house from water damage, and they flow from a single, general principle: that water needs to run off of a house in waterfall fashion. Similarly, while there are many specific security practices, they flow from a small set of well-accepted principles. Understanding the fundamental principles puts you in the best position to implement specific practices where needed in your own projects.

3.1. The Principle of Least Privilege

The *principle of least privilege* states that a user or computer program should be given the least amount of privileges necessary to accomplish a task. A common example in which the principle of least privilege works in the physical world is the use of valet keys. A valet is someone that parks your car for you when you arrive at a hotel or restaurant. Car manufacturers give buyers special valet keys with the purchase of their vehicle. When the car owner pulls up at a hotel or restaurant, she gives the valet key to the valet to park the car. The valet key only allows the valet to start the car and drive it to its parking spot, but does not give the valet access to open the glove compartment or the trunk where valuables may be kept. The idea is to only give the valet access to those resources of the car necessary to get the job of parking the car accomplished.[1]

When you design and use software, you should attempt to employ the same kind of mentality with respect to giving programs just enough permissions for the job that they are required to accomplish. If you are designing or implementing a web server that is only responsible for serving static (read-only) marketing pages to web users, the web server should only be given access to the exact set of files that the web server serves to its clients. The web server should not be given privileges to access the company employee database or any other resource that would allow it to do more than serve the marketing pages. By following this approach, if anyone breaks into the web server, the hope is that the most that the attacker will be able to

1. If you wanted to design an even better valet key system for an automobile, you could limit the number of miles that could be driven with the valet key (but that could introduce other safety issues—for instance, if the car would come to a dead stop upon reaching the limit).

do is read the files that make up the marketing pages, because that is all the web server is able to do. If the web server is configured correctly and not given write access to the files, then you would also expect that the attacker would not be able to deface the web site.[2]

Unfortunately, in practice, web servers are sometimes given unnecessary privileges that allow them access to parts of the file system that they do not need access to, and that also allow them to modify files. Attackers are able to do an excessive amount of damage when they crack into such web servers because of these elevated privileges.

For instance, if the system administrator were to run SimpleWebServer (described in Section 2.4) under the root account,[3] then when clients connect to the web server, they would be able to access all files on the system. You might think that this might not be so bad so long as there are no sensitive documents stored in the web server's directory tree. However, due to a vulnerability in SimpleWebServer, an attacker will be able to access all files on the system! We will now illustrate the vulnerability.

Note that in the `serveFile()` function, SimpleWebServer creates a `FileReader` object to read the file that the user requested in memory. While you would expect that typical filenames specified by users in the `GET` request might look like `/index.html`, `/admin/login.php`, or even `/logs/joe/1.txt`, an attacker might issue `GET` requests that are malicious. For instance, an attacker might issue the following request:

```
GET ../../../../etc/shadow HTTP/1.0
```

Due to the way the `FileReader` constructor works, it will attempt to access the file specified by its string argument relative to the current working directory. As a result, by issuing such a `GET` request, the attacker would be able to traverse up the directory tree to the root directory, and then access a file such as `/etc/shadow`, which, on UNIX, contains a list of all usernames and "encrypted" versions of their passwords. Even though the passwords are "encrypted," an attacker may then attempt to mount a *dictionary attack* against the password file, especially if the password system was not designed well. We will cover dictionary attacks and how to build good password systems in Chapter 9.

To prevent this attack, you need to canonicalize and validate the pathname that the client specifies in the `GET` request. Writing such code can often be tricky business. The following might be a first-cut implementation at a function that checks the path with the goal of preventing the attack:

```
String checkPath (String pathname) throws Exception {
            File target = new File (pathname);
            File cwd = new File (System.getProperty("user.dir"));
            String targetStr = target.getCanonicalPath();
            String cwdStr = cwd.getCanonicalPath();
            if (!targetStr.startsWith(cwdStr))
                        throw new Exception("File Not Found");
```

2. There have been known attacks in which attackers take control of the account used to run the web server and then exploit a vulnerability in the operating system to take control of other accounts that have more privileges. However, if there was only a vulnerability in the web server and not an additional one in the operating system, the least privilege approach would prevent the attacker from being able to obtain additional privileges.

3. A root account is one that gives a system administrator complete access to all aspects of a system.

```
            else
                        return targetStr;
}
```

Then, you just pass a normalized path to the `File` constructor in the `serveFile()` method:

```
fr = new FileReader (checkPath(pathname));
```

The `checkPath()` function first creates a `File` object called `target` that corresponds to the pathname that the user requests. Then, it creates a `File` object called `cwd` that corresponds to the current working directory. (The call to `System.getProperty("user.dir")` is used to retrieve the current working directory.) The `getCanonicalPath()` method is called for each file to normalize the pathnames (i.e., eliminate "`.`," "`..`," and other ambiguities in the pathname).[4] If the canonicalization fails for any reason, `checkPath()` will throw an `IOException`. Finally, the `if` statement checks to see if the target pathname is at the same level or below the current working directory. If not, `checkPath()` throws an exception to prevent the case in which an attacker tries to access a file that is above the web server's current working directory.

The preceding example used the `checkPath()` function to help contain the impact if the web server is run as root. Validating the input in the HTTP request prevents an attacker from being able to access files (including those accessible only by root) that are above the directory from which the web server is run. However, if the web server is run as root, an attacker could still successfully place HTTP requests for files only accessible to root that are in or below the directory from which the web server is run, even when `checkPath()` is used to validate the input in the HTTP request. While `checkPath()` helps contain the damage if the principle of least privilege is ignored, to truly avoid vulnerability, the web server should not be run as root.

3.2. Defense-in-Depth

Defense-in-depth, also referred to as *redundancy*, is the second design principle we will discuss in this chapter. To start with a common real-world example, consider how some banks protect themselves from bank robbers.

3.2.1. Prevent, Detect, Contain, and Recover

The point of defense-in-depth is to not rely on any one defense to achieve security. Multiple mechanisms can help you achieve more security than just one. Some mechanisms (such as the security guards outside the bank) might help *prevent* attacks. In the case of a bank robbery, it is usually quite obvious when the robbery is taking place—but in the world of network security, it may not even be clear when an attack is taking place. As such, some mechanisms might help you *detect* when attacks are taking place. Since it is not always possible to prevent attacks altogether, it is important to deploy mechanisms that help you *manage* or *contain* attacks while they are in progress. In some banks, bank tellers are stationed behind bulletproof glass, which helps contain the effect of a bank robbery by working to spare the lives of the bank tellers in the case that violence breaks out. After an attack takes place, you want to be able to *recover* from the attack, to whatever extent possible. Bank tellers may give the robbers a specially prepared briefcase of cash that will spurt dye on the robber when he opens it. The police

4. Note that `getCanonicalPath()` may not work as expected in the presence of hard links.

will then be able to find the bank robber because the dye can only be removed using special chemicals, which helps create accountability.[5] In addition to dye-laced briefcases, banks take out insurance policies to help deal with the financial loss in case the cash cannot be recovered. A good security system, whether it be for physical banks or software information systems, should employ defense-in-depth, and include mechanisms that help to prevent, detect, manage, and recover from attacks.

3.2.2. Don't Forget Containment and Recovery

Some organizations go overboard on deploying too many prevention and detection measures, and do not focus adequately enough on containment or recovery. For example, some organizations will deploy a firewall and IDS, but will not have appropriate measures in place to deal with security alerts generated by them.

Preventive techniques may not be perfect, and may fail at preventing some malicious acts from taking place. On the Internet, malicious traffic needs to be treated as a fact of life, instead of as an error or exceptional condition. It may take some time to identify and/or detect malicious traffic before the connections with malicious sources can be dropped. In the interim, you need to contain damage that can impact the normal operation of the network.

To highlight the importance of attack containment techniques, consider an analogy between defenses of a distributed computer system and national security defenses. On the morning of September 11, 2001, at the time that the first hijacked airplane hit the north tower of the World Trade Center, our nation's preventive defense mechanisms had already failed. The FBI, CIA, NSA, and INS had failed to identify and/or detain the terrorists who had entered the country and had been training to fly commercial airliners. The hijackers were let through the airport security checkpoints and were allowed to board. When the first airplane hit the tower, the hijackers were already in control of two other planes in the air.

After the first airplane hit the north tower, it was, in fact, unclear as to whether what had just happened was an accident, or whether it was an attack. Indeed, it would take the authorities some time to detect exactly what was going on. And, of course, regardless of whether the incident that had just occurred was an attack, it would take quite some time to recover from the situation, to the extent that such incidents can be recovered from. Immediately after the crash of the first airplane, and while the authorities were in the process of detecting exactly what was going on, efforts were focused on containing the effects of the incident, by saving as many lives as possible. Such containment techniques—whether they be protocols that emergency response teams should follow, the activation of additional secure radio frequencies and communication channels for use by authorities to coordinate life-saving efforts, or possible procedures for emergency scrambling of jet fighters—need to be designed, practiced, tested, and put in place well ahead of any such incident.

In a distributed system, it is also important that once malicious parties have breached the preventive mechanisms, and while the existence, locations, and identities of the malicious actors are in the process of being detected, attack containment techniques be used to minimize the impact of the attack while detection and recovery procedures are executing.

5. If robbers know that they might be given dye-laced cash, this may also serve as a deterrent, or a preventive measure, since the only way to check for dye-laced cash may be to open the briefcase. Why go through the trouble of robbing the bank if they may not be able to get usable cash? At the same time, the dye-laced cash is not a pure recovery measure, since it doesn't help the bank get the money back; it only makes it useless (in the case that real cash is in the briefcase).

3.2.3. Password Security Example

To consider an example from the world of password security, system administrators can attempt to prevent password-guessing attacks against their web site by requiring users to choose strong passwords (see Section 9.6). To help detect password-guessing attacks, web server logs can be monitored for a large number of failed logins coming from one or more IP addresses, and mark those IP addresses as suspicious. However, doing that is not enough. It is still likely that the attacker may stumble upon a significant number of valid username and password combinations, and it is important to reduce the number of accounts that get compromised during an attack. One option might be to deny all logins from the suspicious IP addresses to contain the attack, or require an additional check to see if the client presents the web server with a cookie that was provided upon last successful login. (We cover cookies in Section 7.3.)

Still, the attacker may obtain a few valid usernames and passwords, especially if the attacker has access to many IP addresses—but the goal of containment is to lower the impact of the attack, not prevent it entirely. Finally, to recover from the attack, you could monitor account activity for the usernames for which there were successful logins from the suspicious IP addresses, and deny any transactions that look suspicious, such as monetary transfers to addresses outside the country. The web site may also have to file an insurance claim to financially recover from any successful fraud that the attacker was able to conduct, and purchase credit monitoring services for customers whose accounts were compromised.

FAILURES, LIES, AND INFILTRATION

In Appendix A, we provide a framework of security techniques called the failure, lies, and infiltration (FLI) model that can help us provide defense-in-depth by preventing, detecting, containing, and recovering from attacks. It may be useful to view the appendix after reading Chapters 12 and 13 (on cryptography), if you do not have any previous background in that area.

3.3. Diversity-in-Defense

An idea related to defense-in-depth is called diversity-in-defense. Diversity-in-defense is about using multiple heterogeneous systems that do the same thing.

One example of using diversity-in-defense is the use of multiple operating systems within a corporation to mitigate the impact of viruses. For example, one could back up data (say, e-mail) on machines that use different operating systems. If a virus attacks Microsoft Outlook, which only works on Windows platforms, it will be able to corrupt all the computers in a corporation that are running Microsoft Windows. However, it is unlikely that the same virus will be able to attack redundant copies of information stored on machines running a different operating system, such as Linux. Using a variety of operating systems protects the entire corporation against attacks on a particular operating system.

Diversity-in-defense does come at a cost, though. By using more than one OS, the IT staff may come under the burden of having to be experts with more than one technology, and will also have to monitor and apply software patches to multiple technologies. The IT staff must

keep such trade-offs in mind and weigh the extra security that diversity-in-defense might provide against the extra complexity and effort it requires. You can read more about the pros and cons of diversity-in-defense in Dan Geer and Dave Aucsmith's position paper entitled "Monopoly Considered Harmful."

3.4. Securing the Weakest Link

A system is only as strong as its *weakest link*. The weakest link is the part of a system that is the most vulnerable, susceptible, or easiest to attack. In this section, we will discuss some prototypical weak links that may exist in systems.

3.4.1. Weak Passwords

One example of a weak link is something that we mentioned earlier—users having weak passwords. Studies going back to 1979 show that people typically choose weak passwords—for example, in Morris and Thompson's "Password Security: A Case History" (Morris and Thompson 1979), they found that about one-third of their users chose a password that could be found in the dictionary. If an attacker interested in compromising some account in the system tries logging in with a variety of different common usernames and passwords, using words from various dictionaries, he will eventually hit the jackpot with one of them. Password security is such a prevalent problem that we dedicate Chapter 9 to the study of it.

3.4.2. People

Another weak link in the corporate security plan is people. In fact, in our previous example of weak passwords, an employee who chooses a password that is simply her name in reverse could be considered a weak link. Even if an employee chooses a good password, she might get conned by a phone call from the company's "system administrator" asking her for the password. Usually, the bigger the company, the more likely that these types of people-based attacks will work—the larger the company, the more often employees may need to trust people that they don't know in the regular course of their day.

And what about the programmers themselves? No amount of software security techniques will help you if your programmers are malicious! Movies such as *Superman III* and *Office Space* have featured programmers who wrote code that transferred "unnoticed" fractions of cents from banking transactions into their own bank accounts. Malicious programmers can also put back doors into their programs, which can give them control of the system after it is deployed—sometimes such programmers even bill such back doors as "features." Processes can be put in place in which programmers are required to review each other's code prior to incorporating it into a software system, but the problem then usually boils down to how many programmers need to collude to build such surprises into software.

In summary, people often end up being the weak link in many systems. Unfortunately, it is hard to eliminate people because your business typically depends on them! To help deal with such people-related threats, a company should create a culture in which their employees enjoy what they do, believe in the goals of the company, are well compensated, and do not have too many incentives to defraud the company. Even then, it may be in a company's best interest to distribute information on a need-to-know basis, and have employees go through criminal background and other checks upon hire.

3.4.3. Implementation Vulnerabilities

Even a correctly designed piece of software typically has lots of bugs in the implementation of that design. Some of those bugs are likely to lead to exploitable security vulnerabilities. Even though an application might use encryption to protect data, it is often possible for an attacker to get access to the protected data not by attacking the encryption function or cracking the encryption key, but by finding bugs in how the software uses (or rather misuses) the encryption function.

Another common example of implementation vulnerability involves the inadvertent mixing of control and data. Attackers can send input data to a program that gets interpreted as a command, which allows them to take control of the program. Later in the book, we will cover examples of implementation vulnerability–based attacks, such as buffer overflows (Chapter 6) and SQL injection (Chapter 8), as well as solutions to them.

3.5. Fail-Safe Stance

Fail-safe stance involves designing a system in such a way that even if one or more components fail, you can still ensure some level of security. In the physical world, there are many systems that take this type of stance. One example involves how an elevator behaves when the power goes out. When elevators lose power or other types of failures occur, they have the capability to automatically grab and latch onto the cables that support them, or use safeties to grab the guide rails on the sides of the elevator shaft, if necessary. Elevators are designed with the expectation that the power will sometimes fail. Software should similarly be designed with the expectation that things will fail.

For example, a firewall is designed to keep malicious traffic out. If a firewall ever fails, it should deny access by default and not let any traffic in. This will be inconvenient for users, but at least the information system protected by the firewall will not be insecure. If, on the other hand, the firewall fails and decides to let all traffic through, attackers could figure out how to induce the firewall to fail, and then would be able to send malicious traffic in. If the firewall is instead designed to let no traffic in upon failure, attackers would not have any additional incentive (besides that of conducting a DoS attack) to try to get the firewall to fail.

3.5.1. SimpleWebServer Fail-Safe Example

We now show that the implementation of the `serveFile()` method from the previous chapter takes a fail-safe stance. The implementation of `serveFile()` is repeated in the following code for convenience:

```
85          public void serveFile (OutputStreamWriter osw,
86                                 String pathname) throws Exception {
87                  FileReader fr = null;
88                  int c = -1;
89                  StringBuffer sb = new StringBuffer();
90
91                  /* Remove the initial slash at the beginning
92                     of the pathname in the request. */
93                  if (pathname.charAt(0) == '/')
94                          pathname = pathname.substring(1);
```

```
95
96                  /* If there was no filename specified by the
97                     client, serve the "index.html" file. */
98                  if (pathname.equals(""))
99                          pathname = "index.html";
100
101                 /* Try to open file specified by pathname. */
102                 try {
103                         fr = new FileReader (pathname);
104                         c = fr.read();
105                 }
106                 catch (Exception e) {
107                         /* If the file is not found, return the
108                            appropriate HTTP response code. */
109                         osw.write ("HTTP/1.0 404 Not Found\n\n");
110                         return;
111                 }
112
113                 /* If the requested file can be successfully opened
114                    and read, then return an OK response code and
115                    send the contents of the file. */
116                 osw.write ("HTTP/1.0 200 OK\n\n");
117                 while (c != -1) {
118                         sb.append((char)c);
119                         c = fr.read();
120                 }
121                 osw.write (sb.toString());
122         }
```

SimpleWebServer takes a fail-safe stance. If an attacker can force the web server to run out of memory, it will crash, but it will not do something insecure such as skipping an access control check or serving any document requested. How can the attacker force the web server to run out of memory?

Note that the way that the preceding `serveFile()` method works is that it uses a `StringBuffer` object (line 89) to store the contents of the file request prior to sending the data in the file to the client. Lines 117 to 120 load the contents of the file into the `StringBuffer`, and line 121 outputs all the content accumulated by the `StringBuffer` to the `OutputStreamWriter` object. In writing the code as such, the programmer assumes that the file is of finite length, and can be loaded in its entirety before it is sent to the client. Many files are of finite length. However, some files, such as live media streams from a web camera, may not be finite, or should at least be served a little bit at a time instead of all at once.

If the attacker can somehow request an infinite-length file, the contents of the file will be put into the `StringBuffer` until the web server process runs out of memory. While the machine that the web server is running on might not be connected to a web camera, if it is a Linux machine, there is (luckily for the attacker) an infinite-length file that the attacker can use. In Linux, `/dev/random` is a file that returns random bits that could, for example, be used to generate cryptographic keys (see Section 14.2.3). However, an attacker can misuse it as a source of

infinite data. For the moment, let us assume that the `checkPath()` function in Section 3.1 was not implemented. If the attacker connects to the web server and issues `GET //dev/random HTTP/1.0` as an HTTP request, SimpleWebServer will continuously read data from `/dev/random` until the web server runs out of memory and crashes. Even though the web server takes a fail-safe stance and crashes when it runs out of memory, it is important to deal with this bug, as it can be used to conduct a DoS attack.

3.5.2. Attempted Fix 1: Checking the File Length

One way to attempt to handle the problem would be for the web server to have a default maximum amount of data to read from the file. Prior to reading data from the file, the web server could determine how much memory it has available, and only decide to serve the file if it has enough memory to read it in. The `serveFile()` method can be written as follows to implement such a feature:

```
FileInputStream fr = null;
StringBuffer sb = new StringBuffer();
pathname = checkPath(pathname);
File f = new File (pathname);
if (f.isDirectory()) {
            // add list of files in directory
            // to StringBuffer...
}
else {
            if (f.length() > Runtime.getRuntime().freeMemory()) {
                        throw new Exception();
            }
            int c = -1;
            fr = new FileReader (f);
            do {
                        c = fr.read();
                        sb.append ((char)c);
            } while (c != -1);
}
```

Unfortunately, with the preceding approach, while the intentions are in the right place, it will not prevent an attack in which the adversary places an HTTP request for `/dev/random`. The reason the preceding code will not solve the problem is because the operating system will report that the length of the file (`f.length()`) is 0 since `/dev/random` is a special file that does not actually exist on disk.

3.5.3. Attempted Fix 2: Don't Store the File in Memory

An alternate attempt to correct the problem might involve not having SimpleWebServer store the bytes of the file prior to sending it. The following code will stream the bytes of the file incrementally and significantly save memory:

```
FileReader fr = null;
int c = -1;

/* Try to open file specified by pathname */
try {
             fr = new FileReader (pathname);
             c = fr.read();
}
catch (Exception e) {
             /* If the file is not found, return the
                appropriate HTTP response code. */
             osw.write ("HTTP/1.0 404 Not Found");
             return;
}

/* If the requested file can be successfully opened
   and read, then return an OK response code and
   send the contents of the file. */
osw.write ("HTTP/1.0 200 OK");
while (c != -1) {
             osw.write (c);
             c = fr.read();
}
```

However, the problem with the preceding approach is that if the attacker requests `/dev/random`, the server will be forever tied up servicing the attacker's request and will not serve any other legitimate user's request. (Remember, SimpleWebServer is not multithreaded.)

3.5.4. Fix: Don't Store the File in Memory, and Impose a Download Limit

To properly defend against the attack, you can take advantage of the approach in which you do not store the file in memory, and impose a maximum download size. The following code will stream at most `MAX_DOWNLOAD_LIMIT` bytes to the client before returning from `serveFile()`:

```
FileReader fr = null;
int c = -1;
int sentBytes = 0;

/* Try to open file specified by pathname */
try {
             fr = new FileReader (pathname);
             c = fr.read();
}
catch (Exception e) {
             /* If the file is not found, return the
                appropriate HTTP response code. */
```

```
            osw.write ("HTTP/1.0 404 Not Found");
            return;
}

/* If the requested file can be successfully opened
   and read, then return an OK response code and
   send the contents of the file. */
osw.write ("HTTP/1.0 200 OK");
while ( (c != -1) && (sentBytes < MAX_DOWNLOAD_LIMIT) ) {
            osw.write (c);
            sentBytes++;
            c = fr.read();
}
```

If the attacker places an HTTP request for `/dev/random`, the connection to the attacker will be cut off once the server has sent `MAX_DOWNLOAD_LIMIT` bytes of `/dev/random` to the client. While the preceding code will defend against the attack, the downside of the preceding implementation is that a legitimate client can receive a truncated file without any warning or indication. As a result, the downloaded file might be corrupted.

In addition, a DoS attack in which the attacker requests a file such as `/dev/random` will only be somewhat mitigated. We say "somewhat" because if the `MAX_DOWNLOAD_LIMIT` is relatively high, it may be some time before a legitimate client is able to download a file. Hence, it is important to choose a `MAX_DOWNLOAD_LIMIT` that is not so low that legitimate download requests will get cut off, but that is not so high that it will allow abusive requests to tie up the server for too long.

3.6. Secure by Default

When you design a system, it should, by default, be optimized for security wherever possible. One problem that some software vendors have had in the past is that when they deploy their software, they turn on every possible feature, and make every service available to the user by default. From a security standpoint, the more features that are built into a piece of software, the more susceptible it is going to be to an attack. For example, if an attacker is trying to observe the behavior of the application, the more features and functionality one makes available, the more the bad guy can observe. There is a higher probability that the attacker is going to find some potential security vulnerability within any of those given features. A rule of thumb when figuring out what features to make available to the user population by default is that you should only enable the 20 percent of the features that are used by 80 percent of the users. That way, most of the users are very happy with the initial configuration that the software will have. The other 20 percent—the power users—that take advantage of the extra functionality in the product will have to explicitly turn those features on, but that is acceptable because they are the power users anyway, and will not have any problem doing so!

Another related idea you should be familiar with is the term *hardening* a system. An operating system, for instance, can contain a lot of features and functionality when it is shipped by the OS vendor, but the amount of functionality available should be reduced. The reduction involves turning off all unnecessary services by default. For instance, in Section 5.2.1, we describe how a malicious program called the Morris worm took advantage of unhardened

UNIX systems that had an unnecessary "debugging" feature enabled in its mail routing program. The high-level idea here is that because there are more features enabled, there are more potential security exploits. By default, you should turn off as many things as you can and have the default configuration be as secure as it possibly can.

Software vendors have recently started taking the concept of secure defaults much more seriously. For example, the Microsoft Windows operating system was originally deployed with all of its features on in the initial configuration. Microsoft configured various functionality offered by their operating system such that it was enabled by default. However, having Internet Information Sever (IIS), Microsoft's web server, on by default made millions of Microsoft Windows computers easier to attack by malicious parties. Worms such as Code Red and Nimda used exploits in IIS to infect the computer on which it was running, and used it as a launching pad to infect other machines. (We discuss more about worms and how they work in Chapter 5.) Because other computers running Windows had IIS turned on by default (even if the users were not using it), the worm was able to spread and infect the other computers quickly.

In newer versions of Windows, Microsoft has turned IIS, as well as many other features in the operating system, off by default. This drastically reduces the ability of a worm to spread over the network. Code Red and Nimda were able to infect thousands of computers within hours because the IIS web server was on by default. Hardening the initial configuration of Windows is one example of how keeping features off by default helps reduce the security threat posed by worms.

3.7. Simplicity

Keeping software as simple as possible is another way to preserve software security. Complex software is likely to have many more bugs and security holes than simple software. Code should be written so that it is possible to test each function in isolation.

One example of a large, complicated piece of software that has had many security holes is the UNIX sendmail program (`www.sendmail.org`). The sendmail program is installed on many UNIX servers deployed on the Internet, and its goal is to route mail from a sender to a recipient.

The simpler the design of a program and the fewer lines of code, the better. A simpler design and fewer lines of code can mean less complexity, better understandability, and better auditability. That does not mean that you should artificially make code compact and unreadable. It means that you should avoid unnecessary mechanisms in your code in favor of simplicity.

In order to keep software simple and security checks localized, you can take advantage of a concept called a choke point. A *choke point* is a centralized piece of code through which control must pass. You could, for instance, force all security operations in a piece of software to go through one piece of code. For example, you should only have one `checkPassword()` function in your system—all password checks should be centralized, and the code that does the password check should be as small and simple as possible so that it can be easily reviewed for correctness. The advantage is that the system is more likely to be secure as long as the code is correct. This is all built on the concept that the less functionality one has to look at in a given application, the less security exposure and vulnerability that piece of software will have. Software that is simple will be easier to test and keep secure.

3.8. Usability

Usability is also an important design goal. For a software product to be *usable*, its users, with high probability, should be able to accomplish tasks that the software is meant to assist them in carrying out. The way to achieve usable software is not to build a software product first, and then bring in an interaction designer or usability engineer to recommend tweaks to the user interface. Instead, to design usable software products, interaction designers and usability engineers should be brought in at the start of the project to architect the information and task flow to be intuitive to the user.

There are a few items to keep in mind regarding the interaction between usability and security:

- *Do not rely on documentation.* The first item to keep in mind is that users generally will not read the documentation or user manual. If you build security features into the software product and turn them off by default, you can be relatively sure that they will not be turned on, even if you tell users how and why to do so in the documentation.

- *Secure by default.* Unlike many other product features that should be turned off by default, security features should be turned on by default, or else they will rarely be enabled at all. The challenge here is to design security features that are easy enough to use that they provide security advantages, and are not inconvenient to the point that users will shut them off or work around them in some way. For instance, requiring a user to choose a relatively strong but usable password when they first power up a computer, and enter it at the time of each login might be reasonable. However, requiring a user to conduct a two-factor authentication every time that the screen locks will probably result in a feature being disabled or the computer being returned to the manufacturer. If the users attempt to do something that is insecure, and they are unable to perform the insecure action, it will at least prevent them from shooting themselves in the foot. It may encourage them to read the documentation before attempting to conduct a highly sensitive operation. Or it may even encourage them to complain to the manufacturer to make the product easier to use.

- *Remember that users will often ignore security if given the choice.* If you build a security prompt into a software product, such as a dialog box that pops up in front of the users saying, "The action that you are about to conduct may be insecure. Would you like to do it anyway?" a user will most likely ignore it and click "Yes." Therefore, you should employ secure-by-default features that do not allow the user to commit insecure actions. These default features should not bother asking the user's permission to proceed with the potentially insecure action. The usability of the application may be negatively impacted, but it will also lead to better security. It also probably means that the product should be redesigned or refactored to assist users in carrying out the task they seek to accomplish in a more secure fashion. Remember that if users are denied the ability to carry out their work due to security restrictions, they will eventually find a way to work around the software, and that could create an insecure situation in itself. The balance between usability and security should be carefully maintained.

The usability challenge for security software products seems to be greater than for other types of products. In a seminal paper entitled "Why Johnny Can't Encrypt," Alma Whitten and Doug Tygar conducted a usability study of PGP (a software product for sending and receiving encrypted e-mail) and concluded that most users were not able to successfully send or receive encrypted e-mail, even if the user interface for the product seemed "reasonable." Even worse, many of the users in their tests conducted actions that compromised the security of the sensitive e-mail with which they were tasked to send and receive. Whitten and Tygar concluded that a more particular notion of "usability for security" was important to consider in the design of the product if it were to be both usable and secure (Whitten and Tygar 1999).

Quoting from their paper, "Security software is usable if the people who are expected to be using it: (1) are reliably made aware of the security tasks they need to perform; (2) are able to figure out how to successfully perform those tasks; (3) don't make dangerous errors; and (4) are sufficiently comfortable with the interface to continue using it."

3.9. Security Features Do Not Imply Security

Using one or more security features in a product does not ensure security. For example, suppose a password is to be sent from a client to the server, and you do not want an attacker to be able to eavesdrop and see the password during transmission. You can take advantage of a security feature (say, encryption) to encrypt the password at the client before sending it to the server. If the attacker eavesdrops, what she will see is encrypted bits. Yet, taking advantage of a security feature, namely encryption, does not ensure that the client/server system is secure, since there are other things that could go wrong. In particular, encrypting the client's password does not ensure protection against weak passwords. The client may choose a password that is too short or easy for the attacker to obtain. Therefore, a system's security is not solely dependent upon the utilization of security features in its design, such as the encryption of passwords, but also depends on how it is used.

Another example involves the interaction between a web client and a web server. You may decide to use SSL. SSL is the Secure Sockets Layer protocol that is used to secure communications between most web browsers and web clients. SSL allows the web client and web server to communicate over an encrypted channel with message integrity in which the client can authenticate the server. (Optionally, the server may also authenticate the client.)

Our SimpleWebServer code can be modified to use an SSL connection instead of a regular one:

```
import java.security.*;
import javax.net.ssl.*;

// ... some code excluded ...

         private static final int PORT = 443;
         private static SSLServerSocket dServerSocket;

         public SimpleWebServer () throws Exception {
                  SSLServerSocketFactory factory =
                        (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
                  dServerSocket = (SSLServerSocket)factory.createServerSocket(PORT);
```

```
// ... some code excluded ...

         public void run () throws Exception {
                  while (true) {
                           /* Wait for a connection from a client. */
                           SSLSocket s = (SSLSocket)dServerSocket.accept();

// ... some code excluded ...

                  }

// ... some code excluded ...
```

■ **Note** Some additional code is also required for the server to read a public key certificate as well as a "private" key in order for it to authenticate itself to clients that connect to it. We discuss certificates and public key cryptography in Chapter 13. The additional code is available from `www.learnsecurity.com/ntk`.

Now, for a client to connect to the server, it would connect to port 443, execute an SSL "handshake" (more information on SSL in Section 15.8), and start exchanging HTTP messages over an authenticated, encrypted channel with message integrity in place. A browser that wants to connect to the server would use a URL such as `https://yourcompany.com`. The *s* in *https* signifies that an SSL connection on port 443, by default, should be used.

You may decide to take advantage of SSL as a security feature in SimpleWebServer, but using SSL does not ensure security. In fact, using SSL in the preceding code does not protect you from all the other threats that we discussed earlier in this chapter (directory traversal attacks, DoS attacks, etc.), even though the client and server might communicate over an SSL connection using this code. Taking advantage of SSL security as a feature may prevent an attacker from being able to snoop on the conversation between the client and server, but it does not necessarily result in overall security, since it does not protect against other possible threats. For instance, if you did not canonicalize the pathname in the HTTP request, an attacker could steal the server's `/etc/shadow` file over the SSL connection. The security of a system cannot be guaranteed simply by utilizing one or more security features.

So, once you have fixed all the implementation vulnerabilities described earlier in this chapter *and* added SSL support to SimpleWebServer, is it finally secure? Probably not.[6] There may very well be a few additional vulnerabilities in the code. We leave it as an exercise to the reader (that's you!) to find the extra vulnerabilities.

6. Actually, there definitely *are* additional vulnerabilities in SimpleWebServer—we are just being facetious.

IS MY CODE SECURE?

In general, you don't really know that any piece of code is actually secure. You either know that it is not secure because you found some security bugs that you have not fixed yet, or it is inconclusive as to whether it is secure. You can say what you have tested for to provide a risk assessment, but that doesn't mean it is 100 percent secure. It turns out that for very small programs, it is sometimes feasible to construct mathematical proofs that the program has certain security properties. But that is mostly of theoretical interest. From a practical standpoint, it is usually impossible to say that a program or software system is secure in any absolute way—it is either insecure or the assessment is inconclusive.

Based on how much testing you have done and what you have tested for, you may be able to provide your management with a *risk assessment*. Generally, the more testing, and the more diverse the testing, the less risky—but all it takes is some discrete hole and all security is blown.

To quote Bruce Schneier, "Security is a process, not a product" (Schneier 2000). Security results not from using a few security features in the design of a product, but from how that product is implemented, tested, maintained, and used.

In a sense, security is similar to quality. It is often hard to design, build, and ship a product, and then attempt to make it high-quality after the fact. The quality of a product is inherent to how it is designed and built, and is evaluated based on its intended use. Such is the case with security.

The bad news about security is that an attacker may often need to find only one flaw or vulnerability to breach security. The designers of a system have a much harder job—they need to design and build to protect against all possible flaws if security is to be achieved. In addition, designing a secure system encompasses much more than incorporating security features into the system. Security features may be able to protect against specific threats, but if the software has bugs, is unreliable, or does not cover all possible corner cases, then the system may not be secure even if it has a number of security features.