3

Security Principles to Live By

Application security must be designed and built into your solutions from the start, and in this chapter I'll focus on how to accomplish this goal by covering tried and tested security principles you should adopt as part of an overall process improvement strategy. I'll discuss security design issues that should be addressed primarily by designers, architects, and program managers. This does not mean that developers and testers should not read this chapter—in fact, developers and testers who understand secure design will create more secure software. Let's get started with a look at some high-level concepts.

SD^3: Secure by Design, by Default, and in Deployment

Our team, the Secure Windows Initiative team, has adopted a simple set of strategies called SD^3—for *secure by design, by default, and in deployment*—to help us achieve our short-term and long-term security goals. We've found that these concepts help shape the development process to deliver secure systems.

Secure by Design

If a system is secure by design, it means you have taken appropriate steps to make sure the overall design of the product is sound from the outset. The steps we recommend development groups take to achieve this include the following:

- Assign a "go-to person" for your security issues. This is the person who signs off on the product being secure. She gets the big bucks for doing so. She is not a scapegoat, but someone who can sit in a meeting and say whether the product is secure enough to ship and, if it's not, what needs to be done to rectify the situation.
- Require training for all personnel. See Chapter 2, "The Proactive Security Development Process," for detailed coverage on this subject.
- Make sure threat models are in place by the time the design phase is complete. I'll discuss threat models in Chapter 4, "Threat Modeling," but you should know that they are useful for determining the application's attack profile and which issues should be remedied.
- Adhere to design and coding guidelines. There are examples of secure design, coding, and testing guidelines in Appendix C, "A Designer's Security Checklist," Appendix D, "A Developer's Security Checklist," and Appendix E, "A Tester's Security Checklist." Note that these are minimum guidelines; you should always strive to exceed them.
- Fix all bugs that deviate from the guidelines as soon as possible. Remember that attackers do not care if the code is old or new. If the code has a flaw, it is flawed, regardless of the code's age.
- Make sure the guidelines evolve. Security threats are not static; you should update the guidelines documents as you learn new vulnerabilities and learn new best practices for mitigating them.
- Develop regression tests for all previously fixed vulnerabilities. This is an example of learning from past mistakes, covered later in this chapter. When a security flaw is discovered, distill the attack code to its simplest form and go look for the other related bugs in other parts of your code.
- Simplify the code, and simplify your security model. This is hard to do, especially if you have a large client base that uses many of your features. However, you should have plans in place to simplify old code by shedding unused and insecure features over time. Code tends to be more chaotic and harder to maintain over time, so the time spent removing old code and making things simpler rather than adding features and fixing bugs is time well spent from a security perspective. Code degeneration is often called *code rot*.
- Perform penetration analysis before you ship. Have people try to break the application. Install test servers, and invite the team and external entities to break it. From my experience, unless the penetration team does nothing other than penetrations and are experts in their field, penetration testing will yield marginal results at best. In

fact, it may have a negative effect if not done properly by giving the development team a false sense of security. The same holds true for "hack-fests" where you invite external people to attack your systems. Typically, they are a waste of time unless you are testing for denial of service issues (because most people attempting to compromise the systems are not too bright and resort to flooding attacks).

Secure by Default

The goal of secure by default is to ship a product that is secure enough out of the box. Some ways to achieve this include these:

- Do not install all features and capabilities by default. Apply only those features used by most of your users, and provide an easy mechanism to enable other features.
- Allow least privilege in your application; don't require your code be used by members of the local or domain administrators group when it does not require such elevated capabilities. This is explained in detail later in this chapter, and there's an entire chapter dedicated to the technical aspects of the subject (Chapter 7, "Running with Least Privilege").
- Apply appropriate protection for resources. Sensitive data and critical resources should be protected from attack. I'll cover this in detail in Chapter 6, "Determining Appropriate Access Control."

Secure in Deployment

Secure in deployment means the system is maintainable once your users install the product. You might create a very well-designed and written application, but if it's hard to deploy and administer, it might be hard to keep the application secure as new threats arise. To achieve the secure in deployment goal, you should follow a few simple guidelines:

- Make sure the application offers a way to administer its security functionality. Obviously, without knowing the security settings and configuration of the application, the administrator cannot know whether the application is secure. This includes the ability to know what level of patching the system is at.
- Create good quality security patches as soon as feasible. If a security vulnerability is found in your code, you must turn around the fix as soon as possible—but not too fast! If you create a fix rapidly, you might make a mistake and introduce more errors, so take care to get the fix right.

- Provide information to the user so that she can understand how to use the system in a secure manner. This could be through online help, documentation, or cues on-screen. This topic is discussed in detail in Chapter 24, "Writing Security Documentation and Error Messages."

Security Principles

The rest of this chapter builds on the SD^3 principles. Remember: security is not something that can be isolated in a certain area of the code. Like performance, scalability, manageability, and code readability, security is a discipline that every software designer, developer, and tester has to know about. After working with a variety of development organizations, we've found that if you keep the following design security principles sacrosanct and employ a sound development process, you can indeed build secure systems:

- Learn from mistakes
- Minimize your attack surface
- Use defense in depth
- Use least privilege
- Employ secure defaults
- Remember that backward compatibility will always give you grief
- Assume external systems are insecure
- Plan on failure
- Fail to a secure mode
- Remember that security features != secure features
- Never depend on security through obscurity alone
- Don't mix code and data
- Fix security issues correctly

Numerous other words of wisdom could be included in this list, but I'll focus on these because we've found them to be among the most useful.

Learn from Mistakes

We've all heard that "what doesn't kill you makes you stronger," but I swear that in the world of software engineering we do not learn from mistakes readily. This is also true in the world of security. Some of my favorite quotations regarding learning from past mistakes include the following:

History is a vast early warning system.
——Norman Cousins (1915–1990),
American editor, writer, and author

Those who cannot remember the past are condemned to repeat it.
——George Santayana (1863–1952),
Spanish-born American philosopher and writer

There is only one thing more painful than learning from experience and that is not learning from experience.
——Archibald McLeish (1892–1982),
American poet

If you find a security problem in your software or learn of one in your competitor's products, learn from the mistake. Ask questions like these:

- How did the security error occur?
- Is the same error replicated in other areas of the code?
- How could we have prevented this error from occurring?
- How do we make sure this kind of error does not happen in the future?
- Do we need to update education or analysis tools?

Approach every bug as a learning opportunity. Unfortunately, in the rush to get products to market, development teams tend to overlook this important step, and so we see the same security blunders occur repeatedly. Failure to learn from a mistake increases the probability that you will make the same costly mistake again.

An important item we instigated at Microsoft is a postmortem phase for security bugs fixed through the Microsoft Security Response Center (*www.microsoft.com/security*). The process starts by filling out a document, which our group analyzes to determine what can be learned. The document includes the following fields:

- Product name
- Product version
- Contact person/people
- Bug database numbers
- Description of vulnerability
- Implication of the vulnerability
- Whether the issue exists in the default installation of the product

- What could designers, developers, or testers have done to prevent this flaw?
- Fix details, including code diffs, if appropriate

As Albert Einstein said, "The only source of knowledge is experience," and learning from previous mistakes is a great way to build up security vulnerability knowledge.

A Hard Lesson

About four years ago, an obscure security bug was found in a product I was close to. Once the fix was made, I asked the product team some questions, including what had caused the mistake. The development lead indicated that the team was too busy to worry about such a petty, time-wasting exercise. During the next year, outside sources found three similar bugs in the product. Each bug took about 100 person-hours to remedy.

I presented this to the new development lead—the previous lead had "moved on"—and pointed out that if four similar issues were found in the space of one year, it would be reasonable to expect more. He agreed, and we spent four hours determining what the core issue was. The issue was simple: some developers had made some incorrect assumptions about the way a function was used. Therefore, we looked for similar instances in the entire code base, found four more, and fixed them all. Next, we added some debug code to the function that would cause the application to stop if the false assumption condition arose. Finally, we sent e-mail to the entire development organization explaining the issue and the steps to take to make sure the issue never occurred again. The entire process took less than 20 person-hours.

The issue is no longer an issue. The same mistake is sometimes made, but the team catches the flaw quickly because of the newly added error-checking code. Finding the root of the issue and spending time to rectify that class of bug would perhaps have made the first development lead far less busy!

Tip As my dad once said to me, "You can make just about any mistake—once. But you'd better make sure you learn from it and not make the same mistake again."

Minimize Your Attack Surface

When you install more code and listen on more network-based protocols, you quickly realize that attackers have more potential points of entry. It's important that you keep these points of entry to a minimum and allow your users to enable functionality as they need it. In Chapter 19, "Security Testing," I'll outline the technical details for calculating the relative attack surface of your product, but at a high level you need to count the following that apply to your application:

- Number of open sockets (TCP and UDP)
- Number of open named pipes
- Number of open remote procedure call (RPC) endpoints
- Number of services
- Number of services running by default
- Number of services running in elevated privileges
- Number of ISAPI filters and applications
- Number of dynamic-content Web pages
- Number of accounts you add to an administrator's group
- Number of files, directories, and registry keys with weak access control lists (ACLs)

Not all of these will apply to your application, and the final tally means nothing unless compared with another version of the same application, but the goal is to reduce the number as much as possible. Also, if you install a service as part of your application and if the service is running by default as SYSTEM, that counts as three! During the various security pushes at Microsoft, we've had a favorite catch phrase for designers, architects, and program managers: "Do whatever it takes to reduce your attack surface."

Employ Secure Defaults

Minimizing attack surface also means defining a secure default installation for your product. Employing secure defaults is one of the most difficult yet important goals for an application developer. You need to choose the appropriate features for your users—hopefully, the feature set is based on user feedback and requirements—and make sure these features are secure. The less often used features should be off by default to reduce potential security exposure. If a feature is not running, it cannot be vulnerable to attack. I generally apply the Pareto Principle, otherwise known as the 80-20 rule: which 20 percent of the product is used by 80 percent of the users? The 20 percent feature set is on by default, and the 80

percent feature set is off by default with simple instructions and menu options for the enabling of features. ("Simply add a DWORD registry value, where the low-order 28 bits are used to denote the settings you want to turn off" is not a simple instruction!) Of course, someone on the team will demand that a rarely used feature be turned on by default. Often you'll find the person has a personal agenda: his mom uses the feature, he designed the feature, or he wrote the feature.

> **Note** There is a downside to turning features off by default: setup programs that rely on your feature might fail if they assume your application is running. Don't use this as an excuse to turn the feature back on. The real fix is to resolve the issue in the dependent program setup tool.

Some time ago I performed a security review for a development tool that was a few months from shipping. The tool had a really cool feature that would install and be enabled by default. After the development team had spent 20 minutes explaining how the feature worked, I summed it up in one sentence: "Anyone can execute arbitrary code on any computer that has this software installed." The team members muttered to one another and then nodded. I said, "That's bad!" and offered some advice about how they could mitigate the issue. But they had little time left in the development cycle to fix the problem, so someone responded, "Why don't we ship with the feature enabled and warn people in the documentation about the security implications of the feature?" I replied, "Why not ship with the feature disabled and inform people in the documentation about how they can enable the feature if they require it?" The team's lead wasn't happy and said, "You know people don't read documentation until they really have to! They will never use our cool feature." I smiled and replied, "Exactly! So what makes you think they'll read the documentation to turn the feature off?" In the end, the team pulled the feature from the product—a good thing because the product was behind schedule!

Another reason for not enabling features by default has nothing to do with security: performance. More features means more memory used; more memory used leads to more disk paging, which leads to performance degradation.

> **Important** As you enable more features by default, you increase the potential for a security violation, so keep the enabled feature set to a minimum. Unless you can argue that your users will be massively inconvenienced by a feature being turned off, keep it off and provide an easy mechanism for enabling the feature if it is required.

Use Defense in Depth

Defense in depth is a straightforward principle: imagine your application is the last component standing and every defensive mechanism protecting you has been destroyed. Now you must protect yourself. For example, if you expect a firewall to protect you, build the system as though the firewall has been compromised.

Let's quickly revisit the castle example from the first chapter. This time, your users are the noble family of a castle in the 1500s, and you are the captain of the army. The bad guys are coming, and you run to the lord of the castle to inform him of the encroaching army and of your faith in your archers, the castle walls, and the castle's moat. The lord is pleased. Two hours later you ask for an audience with the lord and inform him that the marauders have broken the defenses and are inside the outer wall. He asks how you plan to further defend the castle. You answer that you plan to surrender because the bad guys are inside the castle walls. A response like yours doesn't get you far in the armed forces. You don't give up—you keep fighting until all is lost or you're told to stop fighting.

Here's another example, one that's a little more modern. Take a look at a bank. When was the last time you entered a bank to see a bank teller sitting on the floor in a huge room next to a massive pile of money. Never! To get to the big money in a bank requires that you get to the bank vault, which requires that you go through multiple layers of defense. Here are some examples of the defensive layers:

- There is often a guard at the bank's entrance.
- Some banks have time-release doors. As you enter the bank, you walk into a bulletproof glass capsule. The door you entered closes, and after a few seconds the glass door to the bank opens. This means you cannot rush in and rush out. In fact, a teller can lock the doors remotely, trapping a thief as he attempts to exit.
- There are guards inside the bank.
- Numerous closed-circuit cameras monitor the movements of everyone in every corner of the bank.
- Tellers do not have access to the vault. (This is an example of least privilege, which is covered next.)
- The vault itself has multiple layers of defense, such as:
    - **a.** It opens only at certain controlled times.
    - **b.** It's made of very thick metal.
    - **c.** Multiple compartments in the vault require other access means.

Unfortunately, a great deal of software is designed and written in a way that leads to total compromise when a firewall is breached. This is not good enough today. Just because some defensive mechanism has been compromised doesn't give you the right to concede defeat. This is the essence of defense in depth: at some stage you have to defend yourself. Don't rely on other systems to protect you. Put up a fight because software fails, hardware fails, and people fail. People build software, people are flawed, and therefore software is flawed. You must assume that errors will occur that will lead to security vulnerabilities. That means the single layer of defense in front of you will probably be compromised, so what are your plans if it is defeated? Defense in depth helps reduce the likelihood of a single point of failure in the system.

> **Important** Always be prepared to defend your application from attack because the security features defending it might be annihilated. Never give up.

Use Least Privilege

All applications should execute with the least privilege to get the job done and no more. I often analyze products that must be executed in the security context of an administrative account—or, worse, as a service running as the Local System account—when, with some thought, the product designers could have not required such privileged accounts. The reason for running with least privilege is quite simple. If a security vulnerability is found in the code and an attacker can inject code into your process, make the code perform sensitive tasks, or run a Trojan horse or virus, the malicious code will run with the same privileges as the compromised process. If the process is running as an administrator, the malicious code runs as an administrator. This is why we recommend people do not run as a member of the local administrators group on their computers, just in case a virus or some other malicious code executes.

Go on, admit it: you're logged on to your computer as a member of the local administrators group, aren't you? I'm not. I haven't been for over three years, and everything works fine. I write code, I debug code, I send e-mail, I sync with my Pocket PC, I create documentation for an intranet site, and do myriad other things. To do all this, you don't need admin rights, so why run as an admin? (I will admit that when I build a new computer I add myself to the admin group, install all the applications I need, and then promptly remove myself.)

Stepping onto the "Logged On as Admin" Soapbox

If I want to do something special, which requires admin privileges, I either use the *runas* command or create a shortcut on the desktop and check the Run As Different User option (Microsoft Windows 2000) or the Run With Different Credentials option (Windows XP) on the Properties page of the shortcut. When I run the application, I enter my local administrator username and password. That way only the application I'm using runs as an admin. When the application closes, I'm not admin any more. You should try it—you will be much safer from attack!

When you create your application, write down what resources it must access and what special tasks it must perform. Examples of resources include files and registry data; examples of special tasks include the ability to log user accounts on to the system, debug processes, or backup data. Often you'll find you do not require many special privileges or capabilities to get any tasks done. Once you have a list of all your resources, determine what might need to be done with those resources. For example, a user might need to read and write to the resources but not create or delete them. Armed with this information, you can determine whether the user needs to run as an administrator to use your application. The chances are good that she does not.

A common use of least privilege again involves banks. The most valued part of a bank is the vault, but the tellers do not generally have access to the vault. That way an attacker could threaten a teller to access the vault, but the teller simply won't know how to do it.

For a humorous look at the principle of least privilege, refer to "If we don't run as admin, stuff breaks" in Appendix B, "Ridiculous Excuses We've Heard." Also, see Chapter 7 for a full account of how you can often get around requiring dangerous privileges.

Tip If your application fails to run unless the user (or service process identity) is an administrator or the system account, determine why. Chances are good that elevated privileges are unnecessary.

Separation of Privilege

An issue related to using least privilege is support for separation of privilege. This means removing high privilege operations to another process and running

that process with the higher privileges required to perform its tasks. Day-to-day interfaces are executed in a lower privileged process.

In June 2002, a severe exploit in OpenSSH v2.3.1 and v3.3, which ships with versions of Apple Mac OS X, FreeBSD and OpenBSD, was mitigated in v3.3 because it supports separation of privilege by default. The code that contained the vulnerability ran with lower capabilities because the *UsePrivilege-Separation* option was set in *sshd_config*. You can read about the issue at *www.openssh.com/txt/preauth.adv*.

Another example or privilege separation is Microsoft Internet Information Services (IIS) 6, which ships in Windows .NET Server. Unlike IIS 5, it does not execute user code in elevated privileges by default. All user mode HTTP requests are handled by external worker processes (named w3wp.exe) that run under the Network Service account, not under the more privileged Local System account. However, the administration and process management process, inetinfo.exe, which has no direct interface to HTTP requests, runs as Local System.

The Apache Web Server is another example. When it starts up, it starts the main Web server process, httpd, as *root* and then spawns new httpd processes that run as the low privilege *nobody* account to handle the Web requests.

Backward Compatibility Will Always Give You Grief

Backward compatibility is another reason to ship secure products with secure defaults. Imagine your application is in use by many large corporations, companies with thousands, if not tens of thousands, of client computers. A protocol you designed is insecure in some manner. Five years and nine versions later, you make an update to the application with a more secure protocol. However, the protocol is not backward compatible with the old version of the protocol, and any computer that has upgraded to the current protocol will no longer communicate with any other version of your application. The chances are slim indeed that your clients will upgrade their computers anytime soon, especially as some clients will still be using version 1, others version 2, and so on. Hence, the weak version of the protocol lives forever!

One good approach to this problem is to make the versions you'll accept configurable. Some customers will run only the latest version, possibly in a high-risk environment. They prefer not to accept the risk involved with using the older versions of the protocols, or they don't have older clients. These customers should have the ability to determine which versions of a given protocol are enabled for their systems.

> **Tip** Be ready to face many upgrade and backward compatibility issues if you have to change critical features for security reasons.

Backward Incompatibility: SMB Signing and TCP/IP

Consider the following backward compatibility problem at Microsoft. The Server Message Block (SMB) protocol is used by file and print services in Windows and has been used by Microsoft and other vendors since the LAN Manager days of the late 1980s. A newer, more secure version of SMB that employs packet signing has been available since Microsoft Windows NT 4 Service Pack 3 and Windows 98. The updated protocol has two main improvements: it closes "man-in-the-middle" attacks, and it supports message integrity checks, which prevent data-tampering attacks. "Man-in-the-middle" attacks occur when a third party between you and the person with whom you are communicating assumes your identity to monitor, capture, and control your communication. SMB raises the security bar by placing a digital signature in each SMB packet, which is then verified by both the client and the server.

Because of these security benefits, SMB signing is worth enabling. However, when it is enforced, only computers employing SMB signing can communicate with one another when using SMB traffic, which means that potentially all computers in an organization must be upgraded to signed SMB—a nontrivial task. There is the option to attempt SMB signing when communication between two machines is established and to fall back to the less secure unsigned SMB if that communication fails. However, this means that an attacker can force the server to use the less secure SMB rather than signed SMB.

Another example is that of Transmission Control Protocol/Internet Protocol (TCP/IP), which is a notoriously insecure protocol. Internet Protocol Security (IPSec) remedies many of the issues with TCP/IP, but not all servers understand IPSec, so it is not enabled by default. TCP/IP will live for a long time, and TCP/IP attacks will continue because of it.

Assume External Systems Are Insecure

Assuming external systems are insecure is related to defense in depth—the assumption is actually one of your defenses. Consider any data you receive from a system you do not have complete control over to be insecure and a source of attack. This is especially important when accepting input from users. Until you can prove otherwise, all external stimuli have the potential to be an attack.

External servers can also be a potential point of attack. Clients can be redirected in a number of ways to the wrong server. As is covered in more depth in Chapter 15, "Socket Security," the DNS infrastructure we rely on to find the cor-

rect server is not very robust. When writing client-side code, do not make the assumption that you're only dealing with a well-behaved server.

Don't assume that your application will always communicate with an application that limits the commands a user can execute from the user interface or Web-based client portion of your application. Many server attacks take advantage of the ease of sending malicious data to the server by circumventing the client altogether. The same issue exists in the opposite direction, clients compromised by rogue servers.

> **Warning** After reading the next chapter, you'll realize that one product of the decomposition of your application into its key components will be a list of trusted and untrusted data sources. Be very wary of data that flows into your trusted process from an untrusted source. You have been warned!

Plan on Failure

As I've mentioned, stuff fails and stuff breaks. In the case of mechanical equipment, the cause might be wear and tear, and in the case of software and hardware, it might be bugs in the system. Bugs happen—plan on them occurring. Make security contingency plans. What happens if the firewall is breached? What happens if the Web site is defaced? What happens if the application is compromised? The wrong answer is, "It'll never happen!" It's like having an escape plan in case of fire—you hope to never have to put the strategy into practice, but if you do you have a better chance of getting out alive.

> **Tip** Death, taxes, and computer system failure are all inevitable to some degree. Plan for the event.

Fail to a Secure Mode

So, what happens when you do fail? You can fail securely or insecurely. Failing to a secure mode means the application has not disclosed any data that would not be disclosed ordinarily, that the data still cannot be tampered with, and so on. Or you can fail insecurely such that the application discloses more than it should or its data can be tampered with (or worse). The former is the only proposition worth considering—if an attacker knows that he can make your code fail, he can bypass the security mechanisms because your failure mode is insecure.

Also, when you fail, do not issue huge swaths of information explaining why the error occurred. Give the user a little bit of information, enough so that the user knows the request failed, and log the details to some secure log file, such as the Windows event log.

For a microview of insecure failing, look at the following (pseudo)code and see whether you can work out the security flaw:

```
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == ERROR_ACCESS_DENIED) {
    // Security check failed.
    // Inform user that access is denied.
} else {
    // Security check OK.
    // Perform task.
}
```

At first glance, this code looks fine, but what happens if *IsAccessAllowed* fails? For example, what happens if the system runs out of memory, or object handles, when this function is called? The user can execute the privileged task because the function might return an error such as *ERROR_NOT_ENOUGH_MEMORY*.

The correct way to write this code is as follows:

```
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == NO_ERROR) {
    // Secure check OK.
    // Perform task.
} else {
    // Security check failed.
    // Inform user that access is denied.
}
```

In this case, if the call to *IsAccessAllowed* fails for any reason, the user is denied access to the privileged operation.

A list of access rules on a firewall is another example. If a packet does not match a given set of rules, the packet should not be allowed to traverse the firewall; instead, it should be discarded. Otherwise, you can be sure there's a corner case you haven't considered that would allow a malicious packet, or a series of such packets, to pass through the firewall. The administrator should configure firewalls to allow only the packet types deemed acceptable through, and everything else should be rejected.

Another scenario, covered in detail in Chapter 10, "All Input is Evil!" is to filter user input looking for potentially malicious input and rejecting the input if it appears to contain malevolent characters. A potential security vulnerability exists if an attacker can create input that your filter does not catch. Therefore, you should determine what is valid input and reject all other input.

> **More Info** An excellent discussion of failing securely is found in *The Protection of Information in Computer Systems,* by Jerome Saltzer and Michael Schroeder and available at *web.mit.edu/Saltzer/www/publications/protection*.

> **Important** The golden rule when failing securely is to deny by default and allow only once you have verified the conditions to allow.

Remember That Security Features != Secure Features

When giving secure coding and secure design presentations to software development teams, I always include this bullet point on the second or third slide:

Security Features != Secure Features

This has become something of a mantra for the Secure Windows Initiative team. We use it to remember that simply sprinkling some magic security pixie dust on an application does not make it secure. We must all be sure to include the correct features—and to employ the correct features correctly—to defend against attack. It's a waste of time using Secure Socket Layer/Transport Layer Security (SSL/TLS) to protect a system if the client-to-server data stream is not what requires defending. (By the way, one of the best ways to employ correct features correctly is to perform threat modeling, the subject of the next chapter.)

Another reason that security features do not necessarily make for a secure application is that those features are often written by the security-conscious people. So the people writing the secure code are working on security features rather than on the application's core features. (This does not mean the security software is free from security bugs, of course, but chances are good the code is cleaner.)

In short, leave it to threat modeling to determine what the appropriate mitigation techniques should be.

Never Depend on Security Through Obscurity Alone

Always assume that an attacker knows everything that you know—assume the attacker has access to all source code and all designs. Even if this is not true, it is trivially easy for an attacker to determine obscured information. Other parts of this book show many examples of how such information can be found. Obscurity is a useful defense, so long as it is not your only defense. In other

words, it's quite valid to use obscurity as a small part of an overall defense in depth strategy.

Don't Mix Code and Data

Mixing code and data is a thorny issue, and it all started with Lotus 1-2-3 version 2.0 in 1985; users expect highly interactive Web pages and applications. Lotus 1-2-3 was a wildly popular spreadsheet program in the mid-1980s and early 1990s, and what set it apart from any other spreadsheet on the market was its ability to perform custom actions defined by the user. Overnight a huge market of developer wanna-bes made money selling their special macros for the program. The world was changed forever. Nevertheless, data is data is data, and once you add code to the data, that "data" becomes dangerous. Look at the number of virus issues that come through e-mail because the e-mail message mixes data (the e-mail message) and code (in the form of script and attachments). Or look at Web page security issues, such as cross-site scripting flaws, that exist because HTML data and JavaScript code are commingled. Don't get me wrong: merging code and data is extraordinarily powerful, but the reality is that the combination of code and data will lead to security exploits.

If your application supports mixing code and data, you should default to not allowing code to execute and to allow the user to determine the policy. This is the default today in Microsoft Office XP. Macros do not run whatsoever, and the user decides whether he will allow macro code to execute.

Fix Security Issues Correctly

If you find a security code bug or a design issue, fix it and go looking for similar issues in other parts of the application. You will find more like it. Security flaws are like cockroaches: you see one in the kitchen, so you get rid of it. The problem is that the creature has many brothers, sisters, grandkids, cousins, nieces, nephews, and so on. If you have a cockroach, you have a cockroach problem! Unfortunately, the same holds true with security bugs—the person writing the code probably made the same mistake elsewhere.

> **Tip** If you find a security code bug or a design issue, fix it and go looking for similar issues in other parts of the application. You will find more like it.

In a similar vein, if you encounter a common flaw pattern, take steps to add defensive mechanisms that reduce the class of issues, don't merely resolve the issues in a piece-meal fashion.

Next, when you make a fix, do so in an open manner. If you really fixed three bugs and not just the one found by the researcher, say so! In my opinion, this shows you care and understand the issues. Covering up security bugs leads to conspiracy theories! That said, be prudent—don't give so much detail that an attacker can compromise unpatched systems. My favorite quote regarding this point is

Conceal a flaw, and the world will imagine the worst.
——Marcus Valerius Martialis,
Roman poet (C. 40 A. D.–C. 104 A. D.)

If you find a security bug, make the fix as close as possible to the location of the vulnerability. For example, if there is a bug in a function named *ProcessData*, make the fix in that function or as close to the function as feasible. Don't make the fix in some faraway code that eventually calls *ProcessData*. If an attacker can circumvent the system and call *ProcessData* directly, or can bypass your code change, the system is still vulnerable to attack.

Finally, if there is a fundamental reason why a security flaw exists, fix the root of the problem. Don't patch it over. Over time patchwork fixes become bigger problems because they often introduce regression errors. As the saying goes, "Cure the problem, not the symptoms."

Summary

In this chapter, I outlined some of the core principles you should adopt when building software today. In my experience, none of these principles are hard to implement, yet the rewards are huge. You should adopt each of these concepts within your development organization as soon as possible. If you had to choose one principle to get you started, choose "Employ secure defaults" because doing so will reduce the potential attack population (and it leads nicely to "Use defense in depth" and "Use least privilege"). A close second would be "Learn from mistakes." It's all very well making a mistake—we're human, and we make mistakes. Just don't keep making the same mistakes!