CHAPTER

18

# Network Attack and Defense

*Whoever thinks his problem can be solved using cryptography, doesn't understand his problem and doesn't understand cryptography.*

—ATTRIBUTED BY ROGER NEEDHAM AND BUTLER LAMPSON TO EACH OTHER

## 18.1 Introduction

Internet security is a fashionable and fast-moving field; the attacks that are catching the headlines can change significantly from one year to the next. Regardless of whether they're directly relevant to the work you do, network-based attacks are so high-profile that they are likely to have some impact, even if you only use hacker stories to get your client to allocate increased budgets to counter the more serious threats. The point is, some knowledge of the subject is essential for the working security engineer.

There are several fashionable ideas, such as that networks can be secured by encryption and that networks can be secured by firewalls. The best place to start debunking these notions may be to look at the most common attacks. (Of course, many attacks are presented in the media as network hacking when they are actually done in more traditional ways. A topical example is the leak of embarrassing emails that appeared to come from the office of the U.K. prime minister, and were initially blamed on hackers. As it turned out, the emails had been fished out of the trash at the home of his personal pollster by a private detective called Benji the Binman, who achieved instant celebrity status [520].)

### 18.1.1 The Most Common Attacks

Many actual attacks involve combinations of vulnerabilities. Examples of vulnerabilities we've seen in earlier chapters include stack overflow attacks (where you pass an

over-long parameter to a program that carelessly executes part of it) and password guessing, both of which were used by the Internet worm. A common strategy is to get an account on any machine on a target network, then install a password sniffer to get an account on the target machine, then use a stack overflow to upgrade to a root account.

The exact vulnerabilities in use change from one year to the next, as bugs in old software get fixed and new software releases a new crop of them. Still, there are some patterns, and some old favorites that keep coming back in new guises. Here's a list of the top 10 vulnerabilities, as of June 2000 [670].

1. *A stack overflow attack on the BIND program*, used by many Unix and Linux hosts for DNS, giving immediate account access.
2. *Vulnerable CGI programs on Web servers*, often supplied by the vendor as sample programs and not removed. CGI program flaws are the common means of taking over and defacing Web servers.
3. *A stack overflow attack on the remote procedure call (RPC) mechanism*, used by many Unix and Linux hosts to support local networking, and which allows intruders immediate account access (this was used by most of the distributed denial of service attacks launched during 1999 and early 2000).
4. *A bug in Microsoft's Internet Information Server (IIS) Web server software*, which allowed immediate access to an administrator account on the server.
5. *A bug in* `sendmail`, *the most common mail program on Unix and Linux computers*. Many bugs have been found in `sendmail` over the years, going back to the very first advisory issued by CERT in 1988. One of the recent flaws can be used to instruct the victim machine to mail its password file to the attacker, who can then try to crack it.
6. *A stack overflow attack on Sun's Solaris operating system*, which allows intruders immediate root access.
7. *Attacks on NFS (which I'll describe shortly) and their equivalents on Windows NT and Macintosh operating systems.* These mechanisms are used to share files on a local network.
8. *Guesses of usernames and passwords*, especially where the root or administrator password is weak, or where a system is shipped with default passwords that people don't bother to change.
9. *The IMAP and POP protocols*, which allow remote access to email but are often misconfigured to allow intruder access.
10. *Weak authentication in the SNMP protocol*, used by network administrators to manage all types of network-connected devices. SNMP uses a default password of "public" (which a few "clever" vendors have changed to "private").

Observe that none of these attacks is stopped by encryption, and not all of them by firewalls. For example, vulnerable Web servers can be kept away from back-end business systems by putting them outside the firewall, but they will still be open to vandalism; and if the firewall runs on top of an operating system with a vulnerability, then the bad guy may simply take it over.

Although some of these attacks may have been fixed by the time this book is published, the underlying pattern is fairly constant. Most of the exploits make use of program bugs, of which the majority are stack overflow vulnerabilities. The exploitation of protocol vulnerabilities (such as NFS) vies with weak passwords for second place.

In effect, there is a race between the attackers, who try to find loopholes, and the vendors, who develop patches for them. Capable motivated attackers may find exploits for themselves and keep quiet about them, but most reported attacks involve exploits that are not only well known but for which tools are available on the Net.

## 18.1.2 Skill Issues: Script Kiddies and Packaged Defense

One of the main culture changes brought by the Net is that, until recently, sophisticated attacks on communications (such as middleperson attacks) were essentially the preserve of national governments. Today, we find not just password-snooping attacks but also more subtle routing attacks being done by kids, for fun. The critical change here is that people write the necessary exploit software, then post it on sites such as `www.rootshell.com`, from which *script kiddies* can download it and use it. This term refers primarily to young pranksters who use attack scripts prepared by others, but it also refers to any unskilled people who download and launch tools they don't fully understand. As systems become ever more complicated, even sophisticated attackers are heading this way; no individual can keep up with all the vulnerabilities that are discovered in operating systems and network protocols. In effect, hacking is being progressively deskilled, while defence is becoming unmanageably complex.

As discussed in Chapter 4, the Internet protocol suite was designed for a world in which trusted hosts at universities and research labs cooperated to manage networking in a cooperative way. That world has passed away. Instead of users being mostly honest and competent, we have a huge user population that's completely incompetent (many of whom have high-speed always-on connections), a (small) minority that's competent and honest, a (smaller) minority that's competent and malicious, and a (less small) minority that's malicious but uses available tools opportunistically.

Deskilling is also a critical factor in defense. There are a few organizations, such as computer companies, major universities, and military intelligence agencies, that have people who know how to track what's going on and tune the defenses appropriately. But most companies rely on a combination of standard products and services. The products include firewalls, virus scanners, and intrusion detection systems; the services are often delivered in the form of new configuration files for these products. In these ways, vulnerabilities become concentrated. An attacker who can work out a defeat of a widely sold system has a wide range of targets to aim at.

We'll now look at a number of specific attack and defense mechanisms. Keep in mind here that the most important attack is the stack overwriting attack, and the second most important is password guessing; but because I already covered the first in Chapter 4 and the second in Chapters 2–3, we'll move down to number three: vulnerabilities in network protocols.

## 18.2 Vulnerabilities in Network Protocols

Commodity operating systems such as Unix and NT are shipped with a very large range of network services, many of which are enabled by default, and/or shipped with configurations that make "plug and play" easy—for the attacker as well as the legitimate user. We will look at both local area and Internet issues; a common theme is that mapping methods (between addresses, filenames, etc.) provide many of the weak points.

This book isn't an appropriate place to explain network protocols, so I offer a telegraphic summary, as follows: the *Internet Protocol* (IP) is a stateless protocol that transfers packet data from one machine to another; it uses 32-bit *IP addresses*, often written as four decimal numbers in the range 0–255, such as 172.16.8.93. Most Internet services use a protocol called *Transmission Control Protocol* (TCP), which is layered on top of IP, and provides virtual circuits by splitting up the data stream into IP packets and reassembling it at the far end, asking for repeats of any lost packets. IP addresses are translated into the familiar Internet host addresses using the *Domain Name System* (DNS), a worldwide distributed service in which higher-level name servers point to local name servers for particular domains. Local networks mostly use Ethernet, in which devices have unique Ethernet addresses, which are mapped to IP addresses using the *Address Resolution Protocol* (ARP).

There are many other components in the protocol suite for managing communications and providing higher-level services. Most of them were developed in the days when the Net had only trusted hosts, and security wasn't a concern. So there is little authentication built in; and attempts to remedy this defect with the introduction of the next generation of IP (IPv6) are likely to take many years.

### 18.2.1 Attacks on Local Networks

Let's suppose that the attacker is one of your employees; he has a machine attached to your LAN, and he wants to take over an account in someone else's name to commit a fraud. Given physical access to the network, he can install packet sniffer software to harvest passwords, get the root password, and create a suitable account. However, if your staff use challenge-response password generators, or are careful enough to only use a root password at the keyboard of the machine it applies to, then he has to be more subtle.

One approach is to try to masquerade as a machine where the target user has already logged on. ARP is one possible target; by running suitable code, the attacker can give wrong answers to ARP messages and claim to be the victim. The victim machine might notice if alert, but the attacker can always wait until it is down—or take it down by using another attack. One possibility is to use *subnet masks*.

Originally, IP addresses used the first 3 bytes to specify the split between the network address and the host address. Now they are interpreted as addressing network, subnetwork, and host, with a variable *network mask*. Diskless workstations, when booting, broadcast a request for a subnet mask; many of them will apply any subnet mask they receive at any time. So by sending a suitable subnet mask, a workstation can be made to vanish.

Another approach, if the company uses Unix systems, is to target Sun's *Network File System* (NFS), the de facto standard for Unix file sharing. This allows a number of workstations to use a network disk drive as if it were a local disk; it has a number of well-known vulnerabilities to attackers who're on the same LAN. When a volume is first mounted, the client requests from the server a *root filehandle*, which refers to the root directory of the mounted file system. This doesn't depend on the time, or the server generation number, and it can't be revoked. There is no mechanism for per-user authentication; the server must trust a client completely or not at all. Also, NFS servers often reply to requests from a different network interface to the one on which the request arrived. So it's possible to wait until an administrator is logged in at a file server, then masquerade as her to overwrite the password file. For this reason, many sites use alternative file systems, such as ANFS.

## 18.2.2 Attacks Using Internet Protocols and Mechanisms

Moving up to the Internet protocol suite, the fundamental problem is similar: there is no real authenticity or confidentiality protection in most mechanisms. This is particularly manifest at the lower-level TCP/IP protocols.

Consider, for example, the three-way handshake used by Alice to initiate a TCP connection to Bob and to set up sequence numbers, shown in Figure 18.1.

This protocol can be exploited in a surprising number of different ways. Now that service denial is becoming really important, let's start off with the simplest service denial attack: the *SYN flood.*

### *18.2.2.1 SYN Flooding*

The SYN flood attack is, simply, to send a large number of SYN packets and never acknowledge any of the replies. This leads the recipient (Bob, in Figure 18.1) to accumulate more records of SYN packets than his software can handle. This attack had been known to be theoretically possible since the 1980s, but came to public attention when it was used to bring down Panix, a New York ISP, for several days in 1996.

A technical fix, the so-called SYNcookie, has been found and incorporated in Linux and some other systems. Rather than keeping a copy of the incoming SYN packet, *B* simply sends out as *Y* an encrypted version of *X.* That way, it's not necessary to retain state about sessions that are half-open.

| | |
|---|---|
| A → B: | SYN; my number is X |
| B → A: | ACK; now X+1 |
| | SYN; my number is Y |
| A → B: | ACK; now Y+1 |
| | (start talking) |

**Figure 18.1** TCP/IP handshake.

#### 18.2.2.2 Smurfing

Another common way of bringing down a host is known as *smurfing*. This exploits the *Internet Control Message Protocol* (ICMP), which enables users to send an echo packet to a remote host to check whether it's alive. The problem arises with broadcast addresses that are shared by a number of hosts. Some implementations of the Internet protocols respond to pings to both the broadcast address and their local address (the idea was to test a LAN to see what's alive). So the protocol allowed both sorts of behavior in routers. A collection of hosts at a broadcast address that responds in this way is called a *smurf amplifier*.

The attack is to construct a packet with the source address forged to be that of the victim, and send it to a number of smurf amplifiers. The machines there will each respond (if alive) by sending a packet to the target, and this can swamp the target with more packets than it can cope with. Smurfing is typically used by someone who wants to take over an *Internet relay chat* (IRC) server, so they can assume control of the chatroom. The innovation was to automatically harness a large number of "innocent" machines on the network to attack the victim.

Part of the countermeasure is technical: a change to the protocol standards in August 1999 so that ping packets sent to a broadcast address are no longer answered [691]. As this gets implemented, the number of smurf amplifiers on the Net is steadily going down. The other part is socioeconomic: sites such as `www.netscan.org` produce lists of smurf amplifiers. Diligent administrators will spot their networks on there and fix them; the lazy ones will find that the bad guys utilize their bandwidth more and more; and thus will be pressured into fixing the problem.

#### 18.2.2.3 Distributed Denial-of-service Attacks

A more recent development along the same lines made its appearance in October 1999. This is the *distributed denial of service* (DDoS) attack. Rather than just exploiting a common misconfiguration as in smurfing, an attacker subverts a large number of machines over a period of time, and installs custom attack software in them. At a predetermined time, or on a given signal, these machines all start to bombard the target site with messages [253]. The subversion may be automated using methods similar to those in the Morris worm.

So far, DDoS attacks have been launched at a number of high-profile Web sites, including Amazon and Yahoo. They could be even more disruptive, as they could target services such as DNS and thus take down the entire Internet. Such an attack might be expected in the event of information warfare; it might also be an act of vandalism by an individual. Curiously, the machines most commonly used as hosts for attack software in early 2000 were U.S. medical sites. They were particularly vulnerable because the FDA insisted that medical Unix machines, when certified for certain purposes, had a known configuration. Once bugs had been discovered in this, there was a guaranteed supply of automatically hackable machines to host the attack software (another example of the dangers of software monoculture).

At the time of writing, the initiative being taken against DDoS attacks is to add *ICMP traceback messages* to the infrastructure. The idea is that whenever a router for-

wards an IP packet, it will also send an ICMP packet to the destination with a probability of about 1 in 20,000. The packet will contain details of the previous hop, the next hop, and as much of the packet as will fit. System administrators will then be able to trace large-scale flooding attacks back to the responsible machines, even when the attackers use forged source IP addresses to cover their tracks [93]. It may also help catch large-scale spammers who abuse *open relays* – relays llowing use by "transit" traffic, that is, messages which neither come from nor go to email addresses for which that SMTP server is intended to provide service.

#### 18.2.2.4 Spam and Address Forgery

Services such as email and the Web (SMTP and HTTP) assume that the lower levels are secure. The most that's commonly done is a look-up of the hostname against an IP address using DNS. So someone who can forge IP addresses can abuse the facilities. The most common example is mail forgery by spammers; there are many others. For example, if an attacker can give DNS incorrect information about the whereabouts of your company's Web page, the page can be redirected to another site—regardless of anything you do, or don't do, at your end. As this often involves feeding false information to locally cached DNS tables, it's called *DNS cache poisoning*.

#### 18.2.2.5 Spoofing Attacks

We can combine some of the preceding ideas into spoofing attacks that work at long range (that is, from outside the local network or domain).

Say that Charlie knows that Alice and Bob are hosts on the target LAN, and wants to masquerade as Alice to Bob. He can take Alice down with a service denial attack of some kind, then initiate a new connection with Bob [559, 90]. This entails guessing the sequence number *Y*, which Bob will assign to the session, under the protocol shown in Figure 18.1. A simple way of guessing *Y*, which worked for a long time, was for Charlie to make a real connection to Alice shortly beforehand and use the fact that the value of *Y* changed in a predictable way between one connection and the next. Modern stacks use random number generators and other techniques to avoid this predictability, but random number generators are often less random than expected—a source of large numbers of security failures [774].

If sequence number guessing is feasible, then Charlie will be able to send messages to Bob, which Bob will believe come from Alice (though Charlie won't be able to read Bob's replies to her). In some cases, Charlie won't even have to attack Alice, just arrange things so that she discards Bob's replies to her as unexpected junk. This is quite a complex attack, but no matter; there are scripts available on the Web that do it.

#### 18.2.2.6 Routing Attacks

Routing attacks come in a variety of flavors. The basic attack involves Charlie telling Alice and Bob that a convenient route between their sites passes through his. Source-level routing was originally introduced into TCP to help get around bad routers. The underlying assumptions—that "hosts are honest" and that the best return path is the best source route—no longer hold, and the only short-term solution is to block source routing. However, it continues to be used for network diagnosis.

Another approach involves redirect messages, which are based on the same false assumption. These effectively say, "You should have sent this message to the other gateway instead," and are generally applied without checking. They can be used to do the same subversion as source-level routing.

Spammers have taught almost everyone that mail forgery is often trivial. Rerouting is harder, since mail routing is based on DNS; but it is getting easier as the number of service providers goes up and their competence goes down. DNS cache poisoning is only one of the tricks that can be used.

## 18.3 Defense against Network Attack

It might seem reasonable to hope that most attacks—at least those launched by script kiddies—can be thwarted by a system administrator who diligently monitors the security bulletins and applies all the vendors' patches promptly to his software. This is part of the broader topic of configuration management.

### 18.3.1 Configuration Management

Tight configuration management is the most critical aspect of a secure network. If you can be sure that all the machines in your organization are running up-to-date copies of the operating system, that all patches are applied as they're shipped, that the service and configuration files don't have any serious holes (such as world-writeable password files), that known default passwords are removed from products as they're installed, and that all this is backed up by suitable organizational discipline, then you can deal with nine and a half of the top ten attacks. (You will still have to take care with application code vulnerabilities such as CGI scripts, but by not running them with administrator privileges you can greatly limit the harm that they might do.)

Configuration management is at least as important as having a reasonable firewall; in fact, given the choice of one of the two, you should forget the firewall. However, it's the harder option for many companies, because it takes real effort as opposed to buying and installing an off-the-shelf product. Doing configuration management by numbers can even make things worse. As noted in Section 18.2.2.3, U.S. hospitals had to use a known configuration, which gave the bad guys a large supply of identically mismanaged targets.

Several tools are available to help the systems administrator keep things tight. Some enable you to do centralized version control, so that patches can be applied overnight, and everything can be kept in synch; others, such as Satan, will try to break into the machines on your network by using a set of common vulnerabilities [320]. Some familiarity with these penetration tools is a very good idea, as they can also be used by the opposition to try to hack you.

The details of the products that are available and what they do change from one year to the next, so it is not appropriate to go into details here. What is appropriate is to say that adhering to a philosophy of having system administrators stop all vulnerabilities at the source requires skill and care; even diligent organizations may find that it is just too expensive to fix all the security holes that were tolerable on a local network but not with an Internet connection. Another problem is that, often, an organisation's most

critical applications run on the least secure machines, as administrators have not dared to apply operating system upgrades and patches for fear of losing service.

This leads us to the use of firewalls.

## 18.3.2 Firewalls

The most widely sold solution to the problems of Internet security is the *firewall.* This is a machine that stands between a local network and the Internet, and filters out traffic that might be harmful. The idea of a "solution in a box" has great appeal to many organizations, and is now so widely accepted that it's seen as an essential part of corporate due diligence. (Many purchasers prefer expensive firewalls to good ones.)

Firewalls come in basically three flavors, depending on whether they filter at the IP packet level, at the TCP session level, or at the application level.

### *18.3.2.1 Packet Filtering*

The simplest kind of firewall merely filters packet addresses and port numbers. This functionality is also available in routers and in Linux. It can block the kind of IP spoofing attack discussed earlier by ensuring that no packet that appears to come from a host on the local network is allowed to enter from outside. It can also stop denial-of-service attacks in which malformed packets are sent to a host, or the host is persuaded to connect to itself (both of which can be a problem for people still running Windows 95).

Basic packet filtering is available as standard in Linux, but, as far as incoming attacks are concerned, it can be defeated by a number of tricks. For example, a packet can be fragmented in such a way that the initial fragment (which passes the firewall's inspection) is overwritten by a subsequent fragment, thereby replacing an address with one that violates the firewall's security policy.

### *18.3.2.2 Circuit Gateways*

More complex firewalls, called *circuit gateways*, reassemble and examine all the packets in each TCP circuit. This is more expensive than simple packet filtering, and can also provide added functionality, such as providing a virtual private network over the Internet by doing encryption from firewall to firewall, and screening out black-listed Web sites or newsgroups (there have been reports of Asian governments building national firewalls for this purpose).

However, circuit-level protection can't prevent attacks at the application level, such as malicious code.

### *18.3.2.3 Application Relays*

The third type of firewall is the *application relay*, which acts as a proxy for one or more services, such as mail, telnet, and Web. It's at this level that you can enforce rules such as stripping out macros from incoming Word documents, and removing active content from Web pages. These can provide very comprehensive protection against a wide range of threats.

The downside is that application relays can turn out to be serious bottlenecks. They can also get in the way of users who want to run the latest applications.

#### 18.3.2.4 Ingress versus Egress Filtering

At present, almost all firewalls point outwards and try to keep bad things out, though there are a few military systems that monitor outgoing traffic to ensure that nothing classified goes out in the clear.

That said, some commercial organizations are starting to monitor outgoing traffic, too. If companies whose machines get used in service denial attacks start getting sued (as has been proposed in [771]), egress packet filtering might at least in principle be used to detect and stop such attacks. Also, as there is a growing trend toward *snitchware*, technology that collects and forwards information about an online subscriber without their authorization. Software that "phones home," ostensibly for copyright enforcement and marketing purposes, can disclose highly sensitive material such as local hard disk directories. I expect that prudent organizations will increasingly want to monitor and control this kind of traffic, too.

#### 18.3.2.5 Combinations

At really paranoid sites, multiple firewalls may be used. There may be a *choke*, or packet filter, connecting the outside world to a screened subnet, also known as a *demilitarized zone* (DMZ), which contains a number of application servers or proxies to filter mail and other services. The DMZ may then be connected to the internal network via a further filter that does network address translation. Within the organization, there may be further boundary control devices, including pumps to separate departments, or networks operating at different clearance levels to ensure that classified information doesn't escape either outward or downward (Figure 18.2).

Such elaborate installations can impose significant operational costs, as many routine messages need to be inspected and passed by hand. This can get in the way so much that people install unauthorized back doors, such as dial-up standalone machines, to get their work done. And if your main controls are aimed at preventing information leaking outward, there may be little to stop a virus getting in. Once in a place it wasn't expected, it can cause serious havoc. I'll discuss this sort of problem in Section 18.4.6 later.

### 18.3.3 Strengths and Limitations of Firewalls

Since firewalls do only a small number of things, it's possible to make them very simple, and to remove many of the complex components from the underlying operating system (such as the RPC and sendmail facilities in Unix). This eliminates a lot of vulnerabilities and sources of error. Organizations are also attracted by the idea of having only a small number of boxes to manage, rather than having to do proper system administration for a large, heterogeneous population of machines.

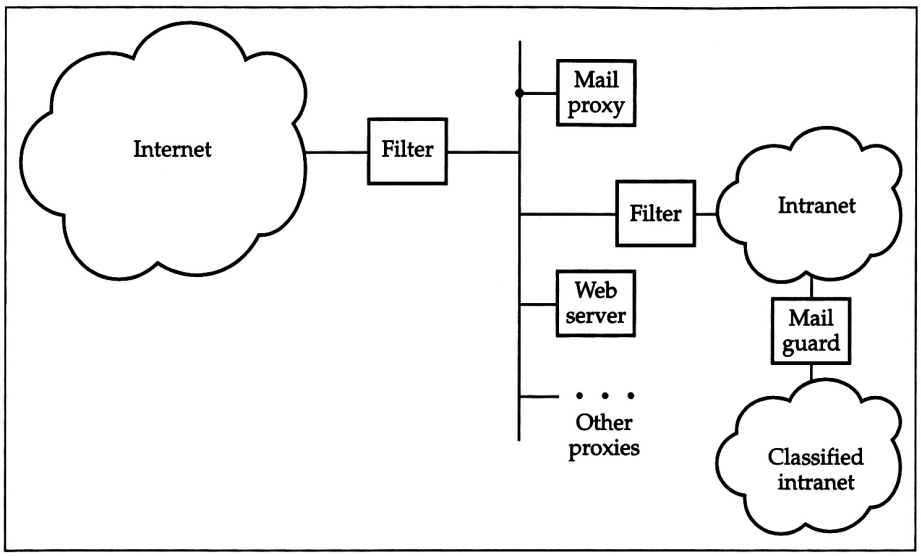


**Figure 18.2** Multiple firewalls.

Conversely, the appeal of simplicity can be seductive and treacherous. A firewall can only be as good as its configuration, and many organizations don't learn enough to do this properly. They hope that by getting the thing out of the box and plugged it in, the problem will be solved. It won't be. It may not require as much effort to manage a firewall as to configure every machine on your network properly in the first place, but it still needs some. In [203], there is a case study of how a firewall was deployed at Hanscom Air Force Base. The work involved the following: surveying the user community to find which network services were needed; devising a network security policy; using network monitors to discover unexpected services that were in use; and lab testing prior to installation. Once it was up and running, the problems included ongoing maintenance (due to personnel turnover), the presence of (unmonitored) communications to other military bases, and the presence of modem pools. Few nonmilitary organizations are likely to take this much care.

A secondary concern, at least during the late 1990s, was that many of the products crowding into the market simply weren't much good. The business had grown so quickly, and so many vendors had climbed in, that the available expertise was spread too thinly.

The big trade-off remains security versus performance. Do you install a simple filtering router, which won't need much maintenance, or do you go for a full-fledged set of application relays on a DMZ, which not only will need constant reconfiguration—as your users demand lots of new services that must pass through it—but will also act as a bottleneck?

An example in Britain was the NHS Network, a private intranet intended for all health service users (family doctors, hospitals, and clinics—a total of 11,000 organizations employing about a million staff in total). Initially, this had a single firewall to the outside world. The designers thought this would be enough, as they expected most traffic to be local (as most of the previous data flows in the health service had been). What they didn't anticipate was that, as the Internet took off in the mid-1990's, 40% of traffic at every level became international. Doctors and nurses found it very convenient to consult medical reference sites, most of which were in America. Trying to squeeze all this traffic through a single orifice was unrealistic. Also, since almost all attacks on healthcare systems come from people who're already inside the system, it was unclear what this central firewall was ever likely to achieve.

Another issue with firewalls (and boundary control devices in general) is that they get in the way of what people want to do, and so ways are found round them. As most firewalls will pass traffic that appears to be Web pages and requests (typically because it's for port 80), more and more applications use port 80, as it's the way to get things to work through the firewall. Where this isn't possible, the solution is for whole services to be reimplemented as Web services (webmail being a good example). These pressures continually erode the effectiveness of firewalls, and bring to mind John Gilmore's famous saying that 'the Internet interprets censorship as damage, and routes around it.'

Finally, it's worth going back down the list of top ten attacks and asking how many of them a firewall can stop. Depending on how it's configured, the realistic answer might be about four.

### 18.3.4 Encryption

In the context of preventing network attacks, many people have been conditioned to think of encryption. Certainly, it can sometimes be useful. For example, on the network at the lab I work in, we use a product called *secure shell* (SSH), which provides encrypted links between Unix and Windows hosts [817, 1, 597]. When I dial in from home, my traffic is protected; and when I log on from the PC at my desk to another machine in the lab, the password I use doesn't go across the LAN in the clear.

Let's stop and analyze what protection this gives me. Novices and policymakers think in terms of wiretaps, but tapping a dial-up modem line is hard now that modems use adaptive echo cancellation. It essentially involves the attacker inserting two back-to-back modems into the link from my house to the lab. So this is a low-probability threat. The risk of password sniffing on our LAN is much higher; it has happened in the past to other departments. Thus, our network encryption is really providing a lower-cost alternative to the use of handheld password generators.

Another approach is to do encryption and/or authentication at the IP layer, which is to be provided in IPv6, and is available as a retrofit for the current IP protocol as IPsec. An assessment of the protocol can be found in [290]; an implementation is described in [782]. IPsec has the potential to stop some network attacks, and to be a useful component in designing robust distributed systems, but it won't be a panacea. Many machines will have to connect to all comers, and if I can become the administrator of your Web

server by smashing the stack, then no amount of encryption or authentication is likely to help you very much. Many other machines will be vulnerable to attacks from inside the network, where computers have been suborned somehow or are operated by dishonest insiders. There will still be problems such as service denial attacks. Also, deployment is likely to take some years.

A third idea is the *virtual private network* (VPN). The idea here is that a number of branches of a company, or a number of companies that trade with each other, arrange for traffic between their sites to be encrypted at their firewalls. This way the Internet can link up their local networks, but without their traffic being exposed to eavesdropping. VPNs also don't stop the bad guys trying to smash the stack of your Web server or sniff passwords from your LAN, but for companies that might be the target of adversarial interest by developed-country governments, it can reduce the exposure to interception of international network traffic. (It must be said, though, that intercepting bulk packet traffic is much harder than many encryption companies claim; and less well-funded adversaries are likely to use different attacks.)

Encryption can also have a downside. One of the more obvious problems is that if encrypted mail and Web pages can get through your firewall, then they can bring all sorts of unpleasant things with them. This brings us to the problem of malicious code.

## 18.4 Trojans, Viruses, and Worms

If this book had been written even five years earlier, malicious code would have merited its own chapter.

Computer security experts have long been aware of the threat from malicious code, or *malware*. The first such programs were *Trojan horses*, named after the horse the Greeks ostensibly left as a gift for the Trojans but that hid soldiers who subsequently opened the gates of Troy to the Greek army. The use of the term for malicious code goes back many years (see the discussion in [493, p. 7].)

There are also *viruses* and *worms*, which are self-propagating malicious programs, and to which I have referred repeatedly in earlier chapters. There is debate about the precise definitions of these three terms: the common usage is that a Trojan horse is a program that does something malicious (such as capturing passwords) when run by an unsuspecting user; a worm is something that replicates; and a virus is a worm that replicates by attaching itself to other programs.

### 18.4.1 Early History of Malicious Code

Malware seems likely to appear whenever a large enough number of users share a computing platform. It goes back at least to the early 1960s. The machines of that era were slow, and their CPU cycles were carefully rationed among different groups of users. Because students were often at the tail of the queue—they invented tricks such as writing computer games with a Trojan horse inside to check whether the program

was running as root, and if so to create an additional privileged account with a known password. By the 1970s, large time-sharing systems at universities were the target of more and more pranks involving Trojans. All sorts of tricks were developed.

In 1984, there appeared a classic paper by Thompson in which he showed that even if the source code for a system were carefully inspected, and known to be free of vulnerabilities, a trapdoor could still be inserted. His trick was to build the trapdoor into the compiler. If this recognized that it was compiling the login program, it would insert a trapdoor such as a master password that would work on any account. Of course, someone might try to stop this by examining the source code for the compiler, and then compiling it again from scratch. So the next step is to see to it that, if the compiler recognizes that it's compiling itself, it inserts the vulnerability even if it's not present in the source. So even if you can buy a system with verifiably secure software for the operating system, applications and tools, the compiler binary can still contain a Trojan. The moral is that you can't trust a system you didn't build completely yourself; vulnerabilities can be inserted at any point in the tool chain [746].

Computer viruses also burst on the scene in 1984, thanks to the thesis work of Fred Cohen. He performed a series of experiments with different operating systems that showed how code could propagate itself from one machine to another, and (as mentioned in Chapter 7) from one compartment of a multilevel system to another. This caused alarm and consternation; and within about three years, the first real, live viruses began to appear "in the wild." Almost all of them were PC viruses, as DOS was the predominant operating system. They spread from one user to another when users shared programs on diskettes or via bulletin boards.

One of the more newsworthy exceptions was the Christmas Card virus, which spread through IBM mainframes in 1987. Like the more recent Love Bug virus, it spread by email, but that was ahead of its time. The next year brought the Internet worm, which alerted the press and the general public to the problem.

## 18.4.2 The Internet Worm

The most famous case of a service denial attack was the Internet worm of November 1988 [263]. This was a program written by Robert Morris Jr which exploited a number of vulnerabilities to spread from one machine to another. Some of these were general (e.g., 432 common passwords were used in a guessing attack, and opportunistic use was made of `.rhosts` files), and others were system specific (problems with `sendmail`, and the `fingerd` bug mentioned in Section 4.4.1). The worm took steps to camouflage itself; it was called `sh` and it encrypted its data strings (albeit with a Caesar cipher).

Morris claimed that this code was not a deliberate attack on the Internet, merely an experiment to see whether his code could replicate from one machine to another. It could. It also had a bug. It should have recognized already infected machines, and not infected them again, but this feature didn't work. The result was a huge volume of communications traffic that completely clogged up the Internet.

Given that the Internet (or, more accurately, its predecessor the ARPANET) had been designed to provide a very high degree of resilience against attacks—up to and including a strategic nuclear strike—it was remarkable that a program written by a student could disable it completely.

What's less often remarked on is that the mess was cleaned up, and normal service was restored within a day or two; that it only affected Berkeley Unix and its derivatives (which may say something about the dangers of the creeping Microsoft

monoculture today); and that people who stayed calm and didn't pull their network connection recovered more quickly, because they could find out what was happening and get the fixes.

## 18.4.3 How Viruses and Worms Work

A virus or worm will typically have two components: a replication mechanism and a payload. A worm simply makes a copy of itself somewhere else when it's run, perhaps by breaking into another system (as the Internet worm did) or by mailing itself as an attachment to the addresses on the infected system's address list (as a number of more recent worms have done). In the days of DOS viruses, the most common way for a virus to replicate was to append itself to an executable file, then patch itself in, so that the execution path jumped to the virus code, then back to the original program.

Among the simplest common viruses were those that infected `.com` 'type executables under DOS. This file format always had code starting at address 0x100, so it was simple for the virus to attach itself to the end of the file and replace the instruction at 0x100 with a jump to its start address. Thus, the viral code would execute whenever the file was run; it would typically look for other, uninfected, `.com` files and infect them. After the virus had done its work, the missing instruction would be executed and control would be returned to the host program.

Given a specific platform, such as DOS, there are usually additional tricks available to the virus writer. For example, if the target system has a file called accounts.exe, it is possible to introduce a file called accounts.com, which DOS will execute first. This is called a *companion virus*. DOS viruses may also attack the boot sector or the partition table; there are even printable viruses, all of whose opcodes are printable ASCII characters, meaning they can even propagate on paper. A number of DOS viruses are examined in detail in [512].

The second component of a virus is the payload. This will usually be activated by a trigger, such as a date, and may then do one or more of a number of bad things:

- Make selective or random changes to the machine's protection state (this is what we worried about with multilevel secure systems).
- Make selective or random changes to user data (e.g., trash the disk).
- Lock the network (e.g., start replicating at maximum speed).
- Steal resources for some nefarious task (e.g., use the CPU for DES keysearch).
- Get your modem to phone a premium-rate number in order to make money from you for a telephone scamster.
- Steal or even publish your data, including your crypto keys.
- Create a backdoor through which its creator can take over your system later, perhaps to launch a distributed denial of service attack.

Until recently, the most damaging payloads were those that leave backdoors for later use, and those that do their damage slowly and imperceptibly. An example of the second are viruses that occasionally swap words in documents or blocks in files; by the time this kind of damage comes to the administrator's attention, all extant generations of backup may be corrupted. Finally, on September 21st 2000 came a report of a virus with a payload that had long been awaited. Swiss bank UBS warned its customers of a

virus that, if it infected their machines, would try to steal the passwords used to access its electronic home banking system.

Various writers have also proposed "benevolent" payloads, such as to perform software upgrades in a company, to enforce licensing terms, or even to roam the world looking for cheap airline tickets (so-called intelligent agents)—though the idea that a commercial Web site owner would enable alien code to execute on their Web server with a view to driving down their prices was always somewhat of a pious hope.

### 18.4.4 The Arms Race

Once viruses and antivirus software companies had both appeared, there was an arms race in which each tried to outwit the other.

As a virus will usually have some means of recognizing itself, so that it does not infect the same file twice, some early antivirus software *immunized* files, by patching in enough of the virus to fool it into thinking that the file was already infected. However, this is not efficient, and won't work at all against a large virus population. The next generation were *scanners*, programs that searched through the early part of each executable file's execution path for a string of bytes known to be from an identified virus.

Virus writers responded in various ways, such as by delaying the entry point of the virus in the host file code, thereby forcing scanners to check the entire filespace for infection; and by specific counterattacks on popular antivirus programs. The most recent evolution was polymorphic viruses. These change their code each time they replicate, to make it harder to write effective scanners. Typically, they are encrypted, but have a small header that contains code to decrypt them. With each replication, the virus re-encrypts itself under a different key; it may also insert a few irrelevant operations into the decryption code, or change the order of instructions where this doesn't matter. The encryption algorithm is often simple and easy to break, but even so is enough to greatly slow down the scanner.

The other main technical approach to virus prevention is the *checksummer*. This is a piece of software that keeps a list of all the authorized executables on the system, together with checksums of the original versions of these files. However, one leading commercial product merely calculates cyclic redundancy checks using two different polynomials—a technique that could easily be defeated by a smart virus writer. Where the checksummer does use a decent algorithm, the main countermeasure is *stealth*, which in this context means that the virus watches out for operating system calls of the kind used by the checksummer and hides itself whenever a check is being done.

### 18.4.5 Recent History

By the late 1980s and early 1990s, PC viruses had become such a problem that they gave rise to a whole industry of antivirus software writers and consultants. Many people thought that this wouldn't last, as the move from DOS to "proper" operating sys-

tems like Windows would solve the problem. Some of the antivirus pioneers even sold their companies: one of them tells his story in [720].

But the spread of interpreted languages has provided even more fertile soil for mischief. There was a brief flurry of publicity about bad Java applets in the late 1990s, as people found ways of penetrating Java implementations in browsers; this raised security awareness considerably [537]. But the main sources of infection at the start of the twenty-first century are the macro languages in Microsoft products such as Word, and the main transmission mechanism is the Internet. An industry analysis claims that the Net "saved" the antivirus industry [423]. Another view is that it was never really under threat, that users will always want to share code and data, and that, in the absence of trustworthy computing platforms, we can expect malware to exploit whichever sharing mechanisms they use. Still another view is that Microsoft is responsible, as it was reckless in incorporating such powerful scripting capabilities in applications such as word processing. As they say, your mileage may vary.

In any case, Word viruses took over as the main source of infection in the United States in 1996, and in other countries shortly afterward [57]. By 2000, macro viruses accounted for almost all incidents of mobile malicious code. A typical macro virus is a macro that copies itself into uninfected word processing documents on the victim's hard disk, and waits to be propagated as users share documents. Some variants also take more active steps to replicate, such as by causing the infected document to be mailed to people in the victim's address book. (There's a discussion of macro viruses in [128], which also points out that stopping them is harder than was the case for DOS viruses, as the Microsoft programming environment is now much less open, less well documented, and complex.)

In passing, it's worth noting that malicious data can also be a problem. An interesting example is related by David Mazières and Frans Kaashoek, who operate an anonymous remailer at MIT. This device decrypts incoming messages from anywhere on the Net, uncompresses them, and acts on them. Someone sent them a series of 25 Mb messages consisting of a single line of text repeated over and over; these compressed very well and so were only small ciphertexts when input; but when uncompressed, they quickly filled up the spool file and crashed the system [531]. There are also attacks on other programs that do decompression such as MPEG decoders. However, the most egregious cases involve not malicious data but malicious code.

## 18.4.6 Antivirus Measures

In theory, defense has become simple: if you filter out Microsoft executables at your firewall, you can stop most of the bad things out there. In practice, life isn't so simple. A large Canadian company with 85,000 staff did just that, but many of their staff had personal accounts at Web-based email services, so when the Love Bug virus came along it got into the company as Web pages, without going through the mail filter at the firewall. The company had configured its mail clients so that each of them had the entire corporate directory in their personal address book. The result was meltdown as 85,000 mail clients all tried to send an email to each of 85,000 addresses.

For a virus infestation to be self-sustaining, it needs to pass an *epidemic threshold*, at which its rate of replication exceeds the rate at which it's removed [452]. This depends not just on the infectivity of the virus itself, but on the number (and proportion) of connected machines that are vulnerable. Epidemic models from medicine can be applied to some extent, though they are limited by the different topology of software intercourse (sharing of software is highly localized), and so predict higher infection rates than are actually observed. One medical lesson that does seem to apply is that the most effective organizational countermeasures are centralized reporting, and response using selective vaccination [453].

In the practical world, this comes down to managerial discipline. In the days of DOS-based file viruses, this meant controlling all software loaded on the organization's machines, and providing a central reporting point for all incidents. Now that viruses arrive primarily in email attachments or as active content in Web pages, it may involve filtering these things out at the firewall, and, seeing to it that users have prudent default settings on their systems—such as disabling active content on browsers and macros in word processing documents.

The nature of the things that users need to be trained to do, or to not do, will change over time as systems and threats evolve. For example, in the mid-1990s, the main tasks were to stop infections coming in via PCs used at home, both for work and for other things (such as kids playing games), and to get staff to "sweep" all incoming email and diskettes for viruses, using standalone scanning software. (An effective way of doing the latter, adopted at a London law firm, was to reward whoever found a virus with a box of chocolates—which would then be invoiced to the company that had sent the infected file). Now that typical antivirus software includes automatic screening and central reporting, the issues are more diffuse, such as training people not to open suspicious email attachments, and having procedures to deal with infected backups. But as with the organic kind of disease, prevention is better than cure; and software hygiene can be integrated with controls on illegal software copying and unauthorized private use of equipment.

## 18.5 Intrusion Detection

The typical antivirus software product is an example of an *intrusion detection system.* In general, it's a good idea to assume that attacks will happen, and it's often cheaper to prevent some attacks and detect the rest than it is to try to prevent everything. The systems used to detect bad things happening are referred to generically as intrusion detection systems. Other examples from earlier chapters are the application-specific mechanisms for detecting mobile phone cloning and fraud by bank tellers. Certain stock markets have installed systems to try to detect insider trading by looking for suspicious patterns of activity. Although they are all performing very similar tasks, their developers don't talk to each other much, and we see the same old wheels being reinvented again and again.

Intrusion detection in corporate and government networks is a fast-growing field of security research; for example, U.S. military funding grew from almost nothing to millions in the last few years of the twentieth century. This growth has been prompted by the realization that many systems make no effective use of log and audit data. In the

case of Sun's operating system Solaris, for example, we found in 1996 that the audit formats were not documented, and tools to read them were not available. The audit facility seemed to have been installed to satisfy the formal checklist requirements of government systems buyers, rather than to perform any useful function. There was the hope that improving this would help system administrators detect attacks, whether after the fact or even when they were still in progress.

## 18.5.1 Types of Intrusion Detection

The simplest intrusion detection methods involve sounding an alarm when a threshold is passed. Three or more failed logons, a credit card expenditure of more than twice the moving average of the last three months, or a mobile phone call lasting more than six hours, might all flag the account in question for attention. More sophisticated systems generally fall into two categories.

The first, *misuse detection systems*, use a model of the likely behavior of an intruder. An example would be a banking system that alarms if a user draws the maximum permitted amount from a cash machine on three successive days. Another would be a Unix intrusion detection system that looked for a user's account being taken over by someone who used the system in a much more sophisticated way; thus an account whose user previously used only simple commands would alarm if the log showed use of a compiler. An alarm might also be triggered by specific actions such as an attempt to download the password file. In general, most misuse detection systems, like antivirus scanners, look for a *signature*, a known characteristic of some particular attack. One of the most general misuse detection signatures is interest in a *honey trap*—something enticing left to attract attention. I mentioned, for example, that some hospitals maintain dummy records with celebrities' names to entrap staff who don't respect medical confidentiality.

The second type of intrusion detection strategy is *anomaly detection*. Such systems attempt the much harder job of looking for anomalous patterns of behavior in the absence of a clear model of the attacker's modus operandi. The hope is to detect attacks that have not been previously recognized and catalogued. Systems of this type often use artificial intelligence techniques—neural networks are particularly fashionable.

The dividing line between misuse and anomaly detection is somewhat blurred. A particularly good borderline case is given by Benford's law, which describes the distribution of digits in random numbers. One might expect that numbers beginning with the digits 1, 2, . . . 9 would be equally common. But, in fact, numbers that come from random natural sources, so that their distribution is independent of the number system in which they're expressed, have a logarithmic distribution: about 30% of decimal numbers start with 1. (In fact, all binary numbers start with 1, if initial zeroes are suppressed.) Crooked clerks who think up numbers to cook the books, or even use random number generators without knowing Benford's law, are often caught using it [529].

## 18.5.2 General Limitations of Intrusion Detection

Some intrusions are really obvious. If what you're worried about is a script kiddie vandalizing your corporate Web site, then the obvious thing to do is to have a machine in

your operations room that fetches the page once a second, displays it, and rings a really loud alarm when it changes. (Make sure you do this via an outside proxy; and don't forget that it's not just your own systems at risk. The kiddie could replace your advertisers' pictures with porn, for example, in which case you'd want to pull the links to them pretty fast.)

In general however, intrusion detection is a difficult problem. Fred Cohen proved that detecting viruses (in the sense of deciding whether a program is going to do something bad) is as hard as the halting problem, meaning we can't ever expect a complete solution [192].

Another fundamental limitation comes from the fact that there are basically two different types of security failure: those that cause an error (which I defined in Section 6.2 to be an incorrect state) and those that don't. An example of the former is a theft from a bank that leaves traces on the audit trail. An example of the latter is an undetected confidentiality failure caused by a radio microphone placed by a foreign intelligence service in your room. The former can be detected (at least in principle, and forgetting for now about the halting problem) by suitable processing of the data available to you. But the latter can't be. It's a good idea to design systems so that as many failures as possible fall into the former category, but it's not always practicable [182].

There's also the matter of definitions. Some intrusion detection systems are configured to block any instances of suspicious behavior, and, in extreme cases, to take down the affected systems. Apart from opening the door to service denial attacks, this turns the intrusion detection system into an access control mechanism. As we've already seen, access control is in general a hard problem, that incorporates all sorts of issues of security policy which people often disagree on or simply get wrong. (The common misconceptions that you can do access control with intrusion detection mechanisms and that all intrusion detection can be done with neural networks together would imply that some neural network bolted on to a LAN could be trained to enforce something like Bell-LaPadula. This seems fatuous.)

I prefer to define an intrusion detection system as one that monitors the logs and draws the attention of authority to suspicious occurrences. This is closer to the way mobile phone operators work. It's also critical in financial investigations; see [658] for a discussion, by a special agent with the U.S. Internal Revenue Service, of what he looks for when trying to trace hidden assets and income streams. A lot hangs on educated suspicion, based on long experience. For example, a $25 utility bill may lead to a $250,000 second house hidden behind a nominee. Building an effective system means having the people, and the machines, each do the part of the job they're best at; and this means getting the machine to do the preliminary filtering.

Then there's the cost of false alarms. For example, I used to go to San Francisco every May, and I got used to the fact that after I'd used my U.K. debit card in an ATM five days in a row, it would stop working. Not only does this upset the customer, but villains quickly learn to exploit it (as do the customers—I just started making sure I got enough dollars out in the first five days to last me the whole trip). As in so many security engineering problems, the trade-off between the fraud rate and the insult rate is the critical one; And, as I noted in Chapter 13, "Biometrics," Section 13.8, we can't expect to improve this trade-off simply by looking at lots of different indicators. In general, we must expect that an opponent will always get past the threshold if he or she is patient enough, and either does the attack very slowly or does a large number of small attacks.

A particularly intractable problem with commercial intrusion detection systems is *redlining*. When insurance companies used claim statistics on postcodes to decide the level of premiums to charge, it was found that many poor and minority areas suffered high premiums or were excluded altogether from coverage. In a number of jurisdictions, this is now illegal. But the problem is much broader. For example, Washington is pushing airlines to bring in systems to profile passengers for terrorism risk, so they can be subjected to more stringent security checks. The American-Arab Anti-Discrimination Committee has reported many incidents where innocent passengers have been harassed by airlines that have implemented some of these recommendations [516].

In general, if you build an intrusion detection system based on data-mining techniques, you are at serious risk of discriminating. If you use neural network techniques, you'll have no way of explaining to a court what the rules underlying your decisions are, so defending yourself could be hard. Opaque rules can also contravene European data protection law, which entities citizens to know the algorithms used to process their personal data.

In general, most fielded intrusion detection systems use a number of different techniques [661]. They tend to draw heavily on knowledge of the application, and to be developed by slow evolution.

## 18.5.3 Specific Problems Detecting Network Attacks

Turning now to the specific problem of detecting network intrusion, the problem is much harder than, say, detecting mobile phone cloning, for a number of reasons. For starters, the available products still don't work very well, with success rates of perhaps 60–80% in laboratory tests and a high false alarm rate. For example, at the time of writing, the U.S. Air Force has so far not detected an intrusion using the systems it has deployed on local networks—although once one is detected by other means, the traces can be found on the logs.

The reasons for the poor performance include the following, in no particular order.

- *The Internet is a very "noisy" environment, not just at the level of content but also at the packet level.* A large amount of random crud arrives at any substantial site, and enough of it can be interpreted as hostile to generate a significant false alarm rate. A survey by Bellovin [89] reports that many bad packets result from software bugs; others are the fault of out-of-date or corrupt DNS data; and some are local packets that escaped, travelled the world, and returned.
- *There are too few attacks.* If there are ten real attacks per million sessions—which is almost certainly an overestimate—then even if the system has a false alarm rate as low as 0.1%, the ratio of false to real alarms will be 100. I talked about similar problems with burglar alarms in Chapter 10; it's also a well known issue for medics running screening programs for diseases such as HIV where the test error exceeds the organism's prevalence in the population. In general, where the signal is so far below the noise, an alarm system is likely to so fatigue the guards that even the genuine alarms get missed.

- *Many network attacks are-specific to particular versions of software, so most of them concern vulnerabilities in old versions.* Thus, a general misuse detection tool must have a large, and constantly changing, library of attack signatures.
- *In many cases, commercial organizations appear to buy intrusion detection systems simply to tick a "due diligence" box.* This is done to satisfy insurers or consultants.
- *Encrypted traffic, such as SSL-encrypted Web sessions, can't easily be subjected to content analysis or filtered for malicious code.* It's theoretically possible to stop the encryption at your firewall, or install a monitoring device with which your users share their confidentiality keys. However, in practice, this can be an absolute tar-pit [3].
- *The issues raised in the context of firewalls largely apply to intrusion detection, too.* You can filter at the packet layer, which is fast but can be defeated by packet fragmentation; you can reconstruct each session, which takes more computation and so is not really suitable for network backbones; or you can examine application data, which is more expensive still, and needs to be constantly updated to cope with the arrival of new applications.

Although the USAF has so far not found an attack using local intrusion detection systems, attacks have been found using network statistics. Histograms are kept of packets by source and destination address and by port. This is a powerful means of detecting *stealthy* attacks, in which the opponent sends one or two packets per day to each of maybe 100,000 hosts. Such attacks would probably never be found using local statistics, and they'd be lost in the noise floor. But when data collection is done over a large network, the suspect source addresses stick out like the proverbial sore thumb.

For all these reasons, it appears unlikely that a single-product solution will do the trick. Future intrusion detection systems are likely to involve the coordination of a number of monitoring mechanisms at different levels, both in the network (backbone, LAN, individual machine) and in the protocol stack (packet, session, and application). This doesn't mean a clean partition in which packet filtering is done in the backbone and application level stuff at proxies; bulk keyword searching might be done on the backbone (as long as IPsec doesn't cause all the traffic to vanish behind a fog of crypto).

## 18.6 Summary

Preventing and detecting attacks that are launched over networks, and particularly over the Internet, is probably the most newsworthy aspect of security engineering. The problem is unlikely to be solved any time soon, as so many different kinds of vulnerability contribute to the attacker's toolkit. Ideally, people would run carefully written code on secure platforms; in real life, this won't always happen. But there is some hope that firewalls can keep out the worst of the attacks, that careful configuration

management can block most of the rest, and that intrusion detection can catch most of the residue that make it through.

Because hacking techniques depend so heavily on the opportunistic exploitation of vulnerabilities introduced accidentally by the major software vendors, they are constantly changing. In this chapter, I concentrated on explaining the basic underlying science (of which there's surprisingly little). Although the Internet has connected hundreds of millions of machines that are running insecure software, and often with no administration to speak of, and scripts to attack common software products have started to be widely distributed, most of the bad things that happen are the same as those that happened a generation ago. The one new thing to have emerged is the distributed denial-of-service attack, which is made possible by the target system's being connected to many hackable machines. Despite all this, the Internet is not a disaster.

Perhaps a suitable analogy for the millions of insecure computers is given by the herds of millions of gnu which once roamed the plains of Africa. The lions could make life hard for any one gnu, but most of them survived for years by taking shelter in numbers. Things were a bit more tense for the very young, the very old, and those who went for the lusher grazing ahead of the herd. The Internet's much the same. There are analogues of the White Hunter, who'll carefully stalk a prime trophy animal; so you need to take special care if anyone might see you in these terms. (If you think that the alarms in the press about 'Evil Hackers Bringing Down the Internet' are somehow equivalent to the hungry peasant with a Kalashnikov, then it may well be worth bearing in mind the even greater destruction done by colonial ranching companies with the capital to fence off the veld in 100,000-acre lots.)

Of course, if you are going for the lusher grazing, or will have to protect high-profile business-critical systems against network attack, then you should read all the hacker Web pages, examine all the hacker software worth looking at, subscribe to the mailing lists, read the advisories, and install the patches. Although hacking has, to a great extent, been deskilled, a similar approach to defense cannot be expected to work more than some of the time, and box-ticking driven by due-diligence concerns isn't likely to achieve more than a modest amount of operational risk reduction.

## Research Problems

In the academic world, research is starting to center on intrusion detection. One interesting theme is to make smarter antivirus products by exploiting analogies with biology. IBM is automating its techniques for identifying and culturing viruses, with a view to shipping its corporate clients a complete "path lab" [452]; Stephanie Forrest and colleagues at the University of New Mexico mimic the immune system by generating a lot of random "antibodies," then removing those that try to "kill" the system's own tissue [302]. How appropriate are such biological analogies? How far can we take them?

## Further Reading

The classic on Internet security was written by Steve Bellovin and Bill Cheswick [94]. Another solid book is by Simson Garfinkel and Eugene Spafford [331], which is a

good reference for the detail of many of the network attacks and system administration issues. An update on firewalls, and a survey of intrusion detection technology, has been written recently by Terry Escamilla [275]. The seminal work on viruses is by Fred Cohen [192], though it was written before macro viruses became the main problem. Java security is discussed by Gary McGraw and Ed Felten [537] and by Li Gong (its quondam architect) [346]. A survey of security incidents on the Internet appears in a thesis by John Howard [392]. Advisories from CERT [199] and bugtraq [144] are also essential reading if you want to keep up with events; and hacker sites such as `www.phrack.com` and (especially) `www.rootshell.com` bear watching.