5

Classes of Attacks

Thus far, we have discussed a number of techniques for attacking systems. Many of these share common characteristics. It is worthwhile categorizing them; the patterns that develop can suggest where protections need to be tightened.

5.1 Stealing Passwords

pedo mellon a minno
(Speak, friend, and enter.)

> "What does it mean by *speak, friend, and enter?*" asked Merry.
>
> "That is plain enough," said Gimli. "If you are a friend, speak the password, and the doors will open, and you can enter."
>
> . . .
>
> "But do not *you* know the word, Gandalf?" asked Boromir in surprise.
>
> "No!" said the wizard. . . . "I do not know the word—yet. But we shall soon see."

Lord of the Rings
—J.R.R. TOLKIEN

The easiest way into a computer is usually through the front door, which is to say, the *login* command. On nearly all systems, a successful login is based on supplying the correct password within a reasonable number of tries.

The history of the generic (even non-UNIX) login program is a series of escalated attacks and defenses: a typical arms race. We can name early systems that stored passwords in the clear in a file. One system's security was based on the secrecy of the name of that password file: it was readable by any who knew its name. The system's security was "protected" by ensuring that the system's directory command would not list that filename. (A system call did return the filename.)

This approach relied on *security by obscurity*. Obscurity is not a bad security tool, though it has received a bad reputation in this regard. After all, what is a cryptographic key but a small, well-designed piece of obscurity. The failure here was the weakness of the obscurity, and the lack of other layers in the defenses.

System bugs are an exciting way to crack a system, but they are not the easiest way to attack. That honor is reserved for a rather mundane feature: user passwords. A high percentage of system penetrations occur because of the failure of the entire password system.

We write "password system" because there are several causes of failure. However, the most common problem is that people tend to pick very bad passwords. Repeated studies have shown that password-guessing is likely to succeed; see, for example, [Klein, 1990] or [Morris and Thompson, 1979]. We are not saying that *everyone* will pick a poor password, but an attacker usually needs only one bad choice.

Password-guessing attacks take two basic forms. The first involves attempts to log in using known or assumed usernames and likely guesses at passwords. This succeeds amazingly often; sites often have account-password pairs such as *field*-`service`, *guest*-`guest`, etc. These pairs often come out of system manuals! The first try may not succeed, nor even the tenth, but all too often, one will work—and once the attacker is in, your major line of defense is gone. Regrettably, few operating systems can resist attacks from the inside.

This approach should not be possible! Users should not be allowed an infinite number of login attempts with bad passwords, failures should be logged, users should be notified of failed login attempts on their accounts, and so on. None of this is new technology, but these things are seldom done, and even more seldom done correctly. Many common mistakes are pointed out in [Grampp and Morris, 1984], but few developers have heeded their advice. Worse yet, much of the existing logging on UNIX systems is in *login* and *su*; other programs that use passwords—*ftpd*, *rexecd*, various screen-locking programs, etc.—do not log failures on most systems. Furthermore, on systems with good logs, the administrators do not check them regularly. Of course, a log of usernames that didn't log in correctly will invariably contain some passwords.

The second way hackers go after passwords is by matching guesses against stolen password files (`/etc/passwd` on UNIX systems). These may be stolen from a system that is already cracked, in which case the attackers will try the cracked passwords on other machines (users tend to reuse passwords), or they may be obtained from a system not yet penetrated. These are called *dictionary attacks*, and they are usually very successful. Make no mistake about it: If your password file falls into enemy hands, there is a very high probability that your machine *will* be compromised. Klein [1990] reports cracking about 25% of the passwords; if that figure is accurate for your machine, and you have just 16 user accounts, there is a 99% chance that at least one of those passwords will be weak.

Cryptography may not help, either, if keys are derived from user-supplied passwords. Experiments with Kerberos [Wu, 1999] show this quite clearly.

A third approach is to tap a legitimate terminal session and log the password used. With this approach, it doesn't matter how good your password is; your account, and probably your system, is compromised.

We can draw several conclusions from this. The first, of course, is that user education in how to choose good passwords is vital. Sadly, although many years have passed since Morris and

How Long Should a Password Be?

It is generally agreed that the former eight-character limit that UNIX systems imposed is inadequate [Feldmeier and Karn, 1990; Leong and Tham, 1991]. But how long should a password be?

Part of the problem with the UNIX system's password-hashing algorithm is that it uses the seven significant bits of each typed character directly as an encryption key. Because the algorithm used (DES; see[NBS, 1977]) permits only 56 bit keys, the limit of eight is derived, not selected. But that begs the question.

The 128 possible combinations of seven bits are not equally probable. Not only do most people avoid using control characters in their passwords, most do not even use characters other than letters. Most folks, in fact, tend to pick passwords composed solely of lowercase letters.

We can characterize the true value of passwords as keys by using *information theory* [Shannon, 1949]. For ordinary English text of 8 letters, the information content is about 2.3 bits per letter, perhaps less [Shannon, 1948, 1951]. We thus have an effective key length of about 19 bits, not 56 bits, for passwords composed of English words.

Some people pick names (their own, their spouse's, their children's, and so on) for passwords. That gives even worse results, because of just how common certain names are. Experiments performed using the AT&T online phone book show that a first name has only about 7.8 bits of information in the whole name. These are very bad choices indeed.

Longer English phrases have a lower information content per letter, on the order of 1.2 to 1.5 bits. Thus, a password of 16 bytes is not as strong as one might guess if words from English phrases are used; there are only about 19 to 24 bits of information there. The situation is improved if the user picks independent words, to about 38 bits. But if users fill up those bytes with combinations of names, we have not helped the situation much.

With the prevalence of password sniffing, passwords shouldn't be used at all, or at least should be cryptographically hidden from dictionary attacks.

Thompson's paper [1979] on the subject, user habits have not improved much. Nor have tightened system restrictions on allowable passwords helped that much, although there have been a number of attempts, e.g., [Spafford, 1992; Bishop, 1992]. Others have tried to enforce password security through retroactive checking [Muffett, 1992]. But perversity always tends toward a maximum, and the hackers only have to win once.

People pick poor passwords—it's human nature. There have been many attempts to force people to pick hard-to-guess passwords [Brand and Makey, 1985], but without much success. It only takes one account to break into a host, and people with small dictionaries have success rates of better than 20% [Klein, 1990]. Large dictionaries can reach tens of megabytes in size. Dictionaries include words and word stems from most written languages. They can include personal information like room number, phone number, hobbies, favorite authors, and so on. Some of this is, quite helpfully, in the password file itself on many machines; others will happily supply it to callers via the *finger* command.

The immediate goal of many network attacks is not so much to break in directly—that is often harder than is popularly supposed—but to grab a password file. Services that we know have been exploited to snatch password files include FTP, TFTP, the mail system, NIS, *rsh*, *finger*, *uucp*, X11, and more. In other words, it's an easy thing for an attacker to do, if the system administrator is careless or unlucky in choice of host system. Defensive measures include great care and a conservative attitude toward software.

If you cannot keep people from choosing bad passwords, it is vital that the password file itself be kept out of enemy hands. This means that one should

- carefully configure the security features for services such as Sun's NIS,
- restrict files available from *tftpd*, and
- avoid putting a genuine `/etc/passwd` file in the anonymous FTP area.

Some UNIX systems provide you with the capability to conceal the hashed passwords from even legitimate users. If your system has this feature (sometimes called a *shadow* or *adjunct* password file), we strongly urge you to take advantage of it. Many other operating systems wisely hash and hide their password files.

A better answer is to get rid of passwords entirely. Token-based authentication is best; at the least, use a one-time password scheme such as *One-Time Password (OTP)* [Haller, 1994; Haller and Metz, 1996]. Again, though, watch out for guessable passphrases.

5.2 Social Engineering

> "We have to boot up the system."
>
> ...
>
> The guard cleared his throat and glanced wistfully at his book. "Booting is not my business. Come back tomorrow."

> "But if we don't boot the system right now, it's going to get hot for us. Overheat. *Muy caliente* and a lot of money."
>
> The guard's pudgy face creased with worry, but he shrugged. "I cannot boot. What can I do?"
>
> "You have the keys, I know. Let us in so we can do it."
>
> The guard blinked resentfully. "I cannot do that," he stated. "It is not permitted."
>
> . . .
>
> "Have you ever seen a computer crash?" he demanded. "It's horrible. All over the floor!"
>
> *Tea with the Black Dragon*
> —R.A. MACAVOY

Of course, the old ways often work the best. Passwords can often be found posted around a terminal or written in documentation next to a keyboard. (This implies physical access, which is not our principle concern in this book.) The social engineering approach usually involves a telephone and some chutzpah, as has happened at AT&T:

> "This is Ken Thompson. Someone called me about a problem with the *ls* command. He'd like me to fix it."
>
> "Oh, OK. What should I do?"
>
> "Just change the password on my login on your machine; it's been a while since I've used it."
>
> "No problem."

There are other approaches as well, such as mail-spoofing. CERT Advisory CA-91:04 (April 18, 1991) warns against messages (purportedly from a system administrator) asking users to run some "test program" that prompts for a password.

Attackers have also been known to send messages like this:

```
From: smb@research.att.com
To: admin@research.att.com
Subject: Visitor

We have a visitor coming next week.  Could you ask your
SA to add a login for her?  Here's her passwd line; use the
same hashed password.
pxf:5bHD/k5k2mTTs:2403:147:Pat:/home/pat:/bin/sh
```

Note that this procedure is flawed even if the note were genuine. If Pat is a visitor, she should not use the same password on our machines as she does on her home machines. At most, this is a useful way to bootstrap her login into existence, but only if you trust her to change her password to something different before someone can take advantage of this. (On the other hand, it does avoid having to send a cleartext password via e-mail. Pay your money and choose your poison.)

Certain actions simply should not be taken without strong authentication. You have to *know* who is making certain requests. The authentication need not be formal, of course. One of us recently "signed" a sensitive mail message by citing the topic of discussion at a recent lunch. In most (but not all) circumstances, an informal "three-way handshake"—a message and a reply, followed by the actual request—will suffice. This is not foolproof: Even a privileged user's account can be penetrated.

For more serious authentication, the cryptographic mail systems described in Chapter 18 are recommended. But remember: No cryptographic system is more secure than the host system on which it is run. The message itself may be protected by a cryptosystem the NSA couldn't break, but if a hacker has booby-trapped the routine that asks for your password, your mail will be neither secure nor authentic.

Sometimes, well-meaning but insufficiently knowledgeable people are responsible for propagating social engineering attacks. Have you ever received e-mail from a friend warning you that, for example, *sulfnbk.exe* is a virus and should be deleted, and that you should warn all of your friends IMMEDIATELY? It's a hoax, and may even damage your machine if you follow the advice. Unfortunately, too many people fall for it—after all, a trusted friend or colleague warned them.

For an insider's account—nay, a former perpetrator's account—of how to perform social engineering, see [Mitnick *et al.*, 2002].

5.3 Bugs and Back Doors

One of the ways the Internet Worm [Spafford, 1989a, 1989b; Eichin and Rochlis, 1989; Rochlis and Eichin, 1989] spread was by sending new code to the *finger* daemon. Naturally, the daemon was not expecting to receive such a thing, and there were no provisions in the protocol for receiving one. But the program did issue a `gets` call, which does not specify a maximum buffer length. The Worm filled the read buffer and more with its own code, and continued on until it had overwritten the return address in `gets`'s stack frame. When the subroutine finally returned, it branched into that buffer and executed the invader's code. The rest is history.

This buffer overrun is called *stack-smashing*, and it is the most common way attackers subvert programs. It takes some care to craft the code because the overwritten characters are machine code for the target host, but many people have done it. The history of computing and the literature is filled with designs to avoid or frustrate buffer overflows. It is not even possible in many computer languages. Some hardware (like the Burroughs machines of old) would not execute code on the stack. In addition, a number of C compilers and libraries use a variety of approaches to frustrate or detect stack-smashing attempts.

Although the particular hole and its easy analogues have long since been fixed by most vendors, the general problem remains: Writing *correct* software seems to be a problem beyond the ability of computer science to solve. Bugs abound.

Secure Computing Standards

What is a secure computer, and how do you know if you have one? Better yet, how do you know if some vendor is selling one?

The U.S. Department of Defense took a stab at this in the early 1980s, with the creation of the so-called *Rainbow Series*. The Rainbow Series was a collection of booklets (each with a distinctively colored cover) on various topics. The most famous was the "Orange Book" [Brand, 1985], which described a set of security levels ranging from D (least secure) to A1. With each increase in level, both the security features and the assurance that they were implemented correctly went up. The definition of "secure" was, in effect, that it satisfied a security model that closely mimicked the DoD's classification system.

But that was one of the problems: DoD's idea of security didn't match what other people wanted. Worse yet, the Orange Book was built on the implicit assumption that the computers in question were 1970s-style time-sharing machines—classified and unclassified programs were to run on the same (expensive) mainframe. Today's computers are much cheaper. Furthermore, the model wouldn't stop viruses from traveling from low security to high security compartments; the intent was to prevent leakage of classified data via overt and covert channels. There was no consideration of networking issues.

The newer standards from other countries were broader in scope. The U.K. issued its "Confidence Levels" in 1989, and the Germans, the French, the Dutch, and the British produced the Information Technology Security Evaluation Criteria document that was published by the European Commission. That, plus the 1993 Canadian Trusted Computer Product Evaluation Criteria, led to the draft Federal Criteria, which in turn gave rise to the Common Criteria [CC, 1999], adopted by ISO.

Apart from the political aspects—Common Criteria evaluations in any country are supposed to be accepted by all of the signatories—the document tries to separate different aspects of security. Thus, apart from assurance being a separate rating scale (one can have a high-assurance system with certain features, or a low-assurance one with the same features), the different functions were separated. Thus, some secure systems can support cryptography and controls on resource utilization, while not worrying about trusted paths. But this means that it's harder to understand exactly what it means for a system to be "secure"—you have to know what it's designed to do as well.

For our purposes, a bug is something in a program that does not meet its specifications. (Whether or not the specifications themselves are correct is discussed later.) They are thus particularly hard to model because, by definition, you do not know which of your assumptions, if any, will fail.

The Orange Book [Brand, 1985] (see the box on page 101) was a set of criteria developed by the Department of Defense to rate the security level of systems. In the case of the Worm, for example, most of the structural safeguards of the Orange Book would have done no good at all. At best, a high-rated system would have confined the breach to a single security level. The Worm was effectively a denial-of-service attack, and it matters little if a multilevel secure computer is brought to its knees by an unclassified process or by a top-secret process. Either way, the system would be useless.

The Orange Book attempts to deal with such issues by focusing on process and assurance requirements for higher rated systems. Thus, the requirements for a B3 rating includes the following statement in Section 3.3.3.1.1:

> The TCB [trusted computing base] shall be designed and structured to use a complete, conceptually simple protection mechanism with precisely defined semantics. This mechanism shall play a central role in enforcing the internal structuring of the TCB and the system. The TCB shall incorporate significant use of layering, abstraction and data hiding. Significant system engineering shall be directed toward minimizing the complexity of the TCB and excluding from the TCB modules that are not protection-critical.

In other words, good software engineering practices are mandated and enforced by the evaluating agency. But as we all know, even the best-engineered systems have bugs.

The Morris Worm and many of its modern-day descendents provide a particularly apt lesson, because they illustrate a vital point: The effect of a bug is not necessarily limited to ill effects or abuses of the particular service involved. Rather, your entire system can be penetrated because of one failed component. There is no perfect defense, of course—no one ever sets out to write buggy code—but there are steps one can take to shift the odds.

The first step in writing network servers is to be very paranoid. The hackers *are* out to get you; you should react accordingly. Don't believe that what is sent is in any way correct or even sensible. Check all input for correctness in every respect. If your program has fixed-size buffers of any sort (and not just the input buffer), make sure they don't overflow. If you use dynamic memory allocation (and that's certainly a good idea), prepare for memory or file system exhaustion, and remember that your recovery strategies may need memory or disk space, too.

Concomitant with this, you need a precisely defined input syntax; you cannot check something for correctness if you do not know what "correct" is. Using compiler-writing tools such as *yacc* or *lex* is a good idea for several reasons, chief among them is that you cannot write down an input grammar if you don't *know* what is legal. You're forced to write down an explicit definition of acceptable input patterns. We have seen far too many programs crash when handed garbage that the author hadn't anticipated. An automated "syntax error" message is a much better outcome.

The next rule is *least privilege*. Do not give network daemons any more power than they need. Very few need to run as the superuser, especially on firewall machines. For example, some portion

of a local mail delivery package needs special privileges, so that it can copy a message sent by one user into another's mailbox; a gateway's mailer, though, does nothing of the sort. Rather, it copies mail from one network port to another, and that is a horse of a different color entirely.

Even servers that *seem* to need privileges often don't, if structured properly. The UNIX FTP server, to cite one glaring example, uses *root* privileges to permit user logins and to be able to bind to port 20 for the data channel. The latter cannot be avoided completely—the protocol does require it—but several possible designs would let a small, simple, and more obviously correct privileged program do that and only that. Similarly, the login problem could be handled by a front end that processes only the `USER` and `PASS` commands, sets up the proper environment, gives up its privileges, and then executes the *unprivileged* program that speaks the rest of the protocol. (See our design in Section 8.7.)

One final note: Don't sacrifice correctness, and verifiable correctness at that, in search of "efficiency." If you think a program needs to be complex, tricky, privileged, or all of the above to save a few nanoseconds, you've probably designed it wrong. Besides, hardware is getting cheaper and faster; your time for cleaning up intrusions, and your users' time for putting up with loss of service, is expensive, and getting more so.

5.4 Authentication Failures

Доверяй но проверяй — "Trust, but verify."

—RUSSIAN PROVERB

Many of the attacks we have described derive from a failure of authentication mechanisms. By this we mean that a mechanism that might have sufficed has somehow been defeated. For example, source-address validation can work, under certain circumstances (e.g., if a firewall screens out forgeries), but hackers can use *rpcbind* to retransmit certain requests. In that case, the ultimate server has been fooled. The message as it appeared to them was indeed of local origin, but its ultimate provenance was elsewhere.

Address-based authentication also fails if the source machine is not trustworthy. PCs are the obvious example. A mechanism that was devised in the days when time-sharing computers were the norm no longer works when individuals can control their own machines. Of course, the usual alternative—ordinary passwords—is no bargain either on a net filled with personal machines; password-sniffing is easy and common.

Sometimes authentication fails because the protocol doesn't carry the right information. Neither TCP nor IP ever identifies the sending user (if indeed such a concept exists on some hosts). Protocols such as X11 and *rsh* must either obtain it on their own or do without (and if they can obtain it, they have to have some secure way of passing it over the network).

Even cryptographic authentication of the source host or user may not suffice. As mentioned earlier, a compromised host cannot perform secure encryption.

5.4.1 Authentication Races

Eavesdroppers can easily pick up a plain password on an unencrypted session, but they may also have a shot at beating some types of one-time password schemes.[1] A susceptible authentication scheme must have a single valid password for the next login, regardless of the source. The next entry in an OTP list (described in Section 7.4) is a good example, and was the first known target of an attack that we describe here.

For this example, we assume that the password contains only digits and is of known length. The attacker initiates ten connections to the desired service. Each connection is waiting for the same unknown password. The valid user connects, and starts typing the correct password. The attack program watches this, and relays the correct characters to its ten connections as they are typed. When only one digit remains to be entered, the program sends a different digit to each of its connections, before the valid user can type the last digit. Because the computer is faster, it wins the race, and one of the connections is validated. These authentication schemes often allow only a single login with each password, so the valid user will be rejected, and will have to try again. Of course, the attacker needs to know the length of the password, but this is usually well-known.

If an attacker can insert himself between the client and server during authentication, he can win an authenticated connection to the host—he relays the challenge to the client and learns the correct answer. An attack on one such protocol is described in [Bellovin and Merritt, 1994].

The authenticator can do a number of things to frustrate this attack [Haller *et al.*, 1998], but they are patches to an intrinsic weakness of the authentication scheme. Challenge/response authentication completely frustrates this attack, because each of the attacker's connections gets a different challenge and requires a different response.

5.5 Protocol Failures

The previous section discussed situations in which everything was working properly, but trustworthy authentication was not possible. Here, we consider the converse: areas where the protocols themselves are buggy or inadequate, thus denying the application the opportunity to do the right thing.

A case in point is the TCP sequence number attack described in Chapter 2. Because of insufficient randomness in the generation of the initial sequence number for a connection, it is possible for an attacker to engage in source-address spoofing. To be fair, TCP's sequence numbers were not intended to defend against malicious attacks. To the extent that address-based authentication is relied on, though, the protocol definition is inadequate. Other protocols that rely on sequence numbers may be vulnerable to the same sort of attack. The list is legion; it includes the DNS and many of the RPC-based protocols.

In the cryptographic world, finding holes in protocols is a popular game. Sometimes, the creators simply made mistakes. More often, the holes arise because of different assumptions. Proving the correctness of cryptographic exchanges is a difficult business and is the subject of

1. See `http://www.tux.org/pub/security/secnet/papers/secureid.pdf`.

much active research. For now, the holes remain, both in academe and—according to various dark hints by Those Who Know—in the real world as well.

Secure protocols must rest on a secure foundation. Consider *ssh*, which is a fine (well, we hope it's fine) protocol for secure remote access. *Ssh* has a feature whereby a user can specify a trusted public key by storing it in a file called `authorized_keys`. Then, if the client knows the private key, the user can log in without having to type a password. In UNIX, this file typically resides in the `.ssh` directory in the user's home directory. Now, consider the case in which someone uses *ssh* to log into a host with NFS-mounted home directories. In that environment, an attacker can spoof the NFS replies to inject a bogus `authorized_keys` file. Therefore, while *ssh* is viewed as a trusted protocol, it fails to be secure in certain reasonably common environments.

The `authorized_keys` file introduces another subtle vulnerability. If a user gets a new account in a new environment, she typically copies all of her important files there from an existing account. It is not unheard of for users to copy their entire `.ssh` directory, so that all of the *ssh* keys are available from the new account. However, the user may not realize that copying the `authorized_keys` file means that this new account can be accessed by any key trusted to access the previous account. While this may appear like a minor nit, it is possible that the new account is more sensitive, and the automatic granting of access through *ssh* may be undesirable.

Note that this is a case of trust being granted by users, not system administrators. That's generally a bad idea.

Another case in point is a protocol failure in the 802.11 wireless data communication standard. Problems with the design of WEP (see Section 2.5) demonstrate that security is difficult to get right, and that engineers who build systems that use cryptography should consult with cryptographers, rather than to try to design something from scratch. This sort of security is a very specialized discipline, not well suited to amateurs.

5.6 Information Leakage

Most protocols give away some information. Often, that is the intent of the person using those services: to gather such information. Welcome to the world of computer spying. The information itself could be the target of commercial espionage agents or it could be desired as an aid to a break-in. The *finger* protocol is one obvious example. Apart from its value to a password-guesser, the information can be used for social engineering. ("Hey, Robin—the battery on my handheld authenticator died out here in East Podunk; I had to borrow an account to send this note. Could you send me the keying information for it?" "Sure, no problem; I knew you were traveling. Thanks for posting your schedule.")

Even such mundane information as phone and office numbers can be helpful. During the Watergate scandal, Woodward and Bernstein used a *Committee to Re-Elect the President* phone book to deduce its organizational structure [Woodward and Bernstein, 1974]. If you're in doubt about what information can be released, check with your corporate security office; they're in the business of saying "no."

In a similar vein, some sites offer access to an online phone book. Such things are convenient, of course, but in the corporate world, they're often considered sensitive. Headhunters love such

things. They find them useful when trying to recruit people with particular skills. Nor is such information entirely benign at universities. Privacy considerations (and often legal strictures) dictate some care about what information can be released. Examples of this are the *Family Educational Rights and Privacy Act (FERPA)* and the EU Privacy Directives.

Another fruitful source of data is the DNS. We have already described the wealth of data that can be gathered from it, ranging from organizational details to target lists. Controlling the outflow is hard; often, the only solution is to limit the externally visible DNS to list gateway machines only.

Sophisticated hackers know this, of course, and don't take you at your word about what machines exist. They do port number and address space scans, looking for interesting services and hidden hosts. The best defense here is a good firewall; if they can't send packets to a machine, it's much less likely to be penetrated.

5.7 Exponential Attacks—Viruses and Worms

Exponential attacks use programs to spread themselves, multiplying their numbers quickly. When the programs travel by themselves, they are *worms.* When they attach to other programs, they are *viruses.* The mathematics of their spread is similar, and the distinction not that important. The epidemiology of such programs is quite similar to biological infectious agents.

These programs succeed by exploiting common bugs or behaviors found in a large population of susceptible programs or users. They can spread around the world within hours, and potentially in a few minutes [Staniford *et al.*, 2002; Rubin, 2001]. They can cause vast economic harm spread over a large community. The Melissa worm clogged the Microsoft-based e-mail in some companies for five days. Various worms have added substantial load to the entire Internet. (Nor is this threat new, or restricted to the Internet. The "IBM Christmas Card virus" clogged IBM's internal bisync network in 1987. See *RISKS Digest,* Vol. 5, Issue 81.)

These programs tend to infect "targets of opportunity," rather than specific individuals or organizations. But their payloads can and do attack popular political and commercial targets.

There are several ways to minimize the chance of getting a virus. By definition, the least popular way is to stay out of the popular monoculture. If you write your own operating system and applications, you are unlikely to be infectible. Microsoft Windows systems have traditionally hosted the vast majority of viruses, which means that Macintosh and UNIX users have suffered less. But this is changing, especially for Linux users. We are now seeing Linux worms, as well as cross-platform worms that can spread through several monocultures, and by direct network access as well as via Web pages and e-mail.

If you don't communicate with an affected host, you can't get the virus. Careful control of network access and the files obtained from foreign sources can greatly reduce the risk of infection. Note that there are also a number of human-propagated viruses, where people forward messages (often containing urban legends) to all of their friends, with instructions to send to all of their friends. These mostly serve as an annoyance. However, they can cause panic in individuals with less computer knowledge. Some contain incorrect messages that the recipient's computer has been infected. In one instance, this was accompanied by instructions to remove a crucial system file. Many people damaged their own computers by following these instructions.

Virus-scanning software is popular and quite effective against known viruses. The software must be updated constantly, as there is an arms race between virus writers and virus detection software companies. The viruses are becoming fantastically effective at hiding their presence and activities. Virus scanners can no longer be content looking for certain strings in the executable code: They have to emulate the code and look for viral behavior. As the viruses get more sophisticated, virus detection software will probably have to take more time examining each file, perhaps eventually taking too long. It is possible that virus writers may eventually be able to make code that cannot be identified in a reasonable amount of time.

Finally, it would be nice to execute only approved, unmodified programs. There are cryptographic technologies than can work here, but the entire approach is tied up with the political furor over copyright protection mechanisms and privacy.

5.8 Denial-of-Service Attacks

> Hello! Hello! Are you there? Hello! I called you up to say hello. I said hello. Can you hear me, Joe?
>
> Oh, no. I can not hear your call. I can not hear your call at all. This is not good and I know why. A mouse has cut the wire. Good-by!
>
> *One Fish, Two Fish, Red Fish, Blue Fish*
> —DR. SEUSS

We've discussed a wide variety of popular attacks on Internet hosts. These attacks rely on such things as protocol weaknesses, programming bugs in servers, and even inappropriately helpful humans. *Denial-of-Service (DOS)* attacks are a different beast. They are the simple overuse of a service—straining software, hardware, or network links beyond their intended capacity. The intent is to shut down or degrade the quality of a service, and that is generally a modest goal.

These attacks are different because they are obvious, not subtle. Shutting down a service should be easy to detect. Though the attack is usually easy to spot, the source of the attack may not be. They often involve generated packets with spoofed, random (and useless) return addresses.

Distributed Denial-of-Service (DDoS) attacks use many hosts on the Internet. More often than not, the participating hosts are unwitting accomplices to the attack, having been compromised in some way and outfitted with some malicious code. DDoS attacks are more difficult to recover from because the attacks come from all over. They are discussed further in Section 5.8.3.

There is no absolute remedy for a denial-of-service attack. As long as there is a public service, the public can abuse it. It is possible to make a perfectly secure site unavailable to the general public for a fair amount of time, and do this anonymously.

It is easy to compute a conservative value for the cost of a DOS attack. If a Web server is down for several days, a business should have a fairly good idea of what that cost them. If it doesn't, it probably didn't have a good business plan for the Web service in the first place.

Companies may try to recover some of these losses through lawsuits, if a culprit can be located. The attack is obvious and easily explained to a jury. This potential may force intermediate parties, such as ISPs, to cooperate more than they might otherwise. Of course, the trouble is finding someone to sue; DDoS attacks are hard to trace back.

5.8.1 Attacks on a Network Link

Network link attacks can range from a simple flood of e-mail (*mail bombing* or *spamming*)[2] to the transmission of packets carefully crafted to crash software on a target host. The attack may fill a disk, swamp a CPU, crash a system, or simply overload a network link.

The crudest attack is to flood a network link. To flood a network link, attackers need only generate more packets than the recipient can handle. Only the destination field of the packet has to be right: the rest can be random (providing the checksum is correct.) It doesn't take that many packets to fill a T1 link: less than 200 KB/second should do it. This can be launched from a single host, providing the connecting network links are a bit faster than the target's.

Several attackers can cooperatively launch an attack that focuses several generators on a target. The traffic from each generator may be low, but the sum of the attacking rates must be greater than the receiver's network link capacity. If the attack is properly coordinated, as in the case of DDoS attacks, hundreds of compromised hosts with slow network connections can flood a target service connected with a high-capacity network link. Posting e-mail addresses to a very popular Web site, such as Slashdot, could result in e-mail flood attacks once spammers obtain the addresses.

5.8.2 Attacking the Network Layer

Many of the worst attacks are made on the network layer—the TCP/IP implementation in the host. These attacks exploit some performance weakness or bug. Given that a typical TCP/IP implementation involves tens of thousands of lines of C code, and runs in privileged space in most computers, it is hard for a developer to debug all possible problems. The edit/compile/reboot cycle is long, and protocols are notoriously hard to debug, especially the error conditions.

The problem can be bad enough under normal usage. It can get much worse when an active adversary is seeking performance holes or even a packet that will crash the host.

Killer and ICMP Packets

There have been rumors around the Internet for years about more potent—i.e., more evil—packets. We have already seen *killer packets* that can tickle a bug and crash a host. These packets may be very large, oddly fragmented, have strange or nonsensical options, or other attributes that test code that isn't used very often (see, for example, CERT Advisory CA-96:26, December 18, 1996, and CERT Advisory CA-00:11, June 9, 2000). Algorithm-savvy attackers can even push programs to perform inefficiently by exploiting weaknesses in queuing or search methods (see the next section for one such case).

Some folks delight in sending bogus ICMP packets to a site, to disrupt its communications. Sometimes these are `Destination Unreachable` messages. Sometimes they are the more confusing—and more deadly—messages that reset the host's subnet mask. (Why, pray tell, do hosts listen to such messages when they've sent no such inquiry?) Other hackers play games with routing protocols, not to penetrate a machine, but to deny it the ability to communicate with its peers.

2. "Spam" should not be confused with the fine meat products of the Hormel Corporation.

SYN Packet Attacks

Of course, some packets hit their targets harder than others. The first well-publicized denial-of-service attack was directed at an ISP, Panix. Panix received about 150 TCP SYN packets a second (see Section 2.1.3). These packets flooded the UNIX kernel's "half open" connection processing, which was fairly simplistic. When the half-open table was full, all further connection attempts were dropped, denying valid users access to the host. SYN packet attacks are described in some detail in [Northcutt and Novak, 2000].

This is the only attack we didn't document in the first edition of this book, because we had no suggestions for fighting it. The description was removed just before the book went to press, a decision we regret. The Panix attack was made using software that two hacker magazines had published a few months before [daemon9 et al.*, 1996].*

The TCP code in most systems was never designed with such attacks in mind, which is how a fairly slow packet rate could shut down a specific TCP service on a host. These were potent packets against weak software. In the aftermath of the attack, the relevant TCP software was beefed up considerably. All it took was sufficient attention.

Application-Level Attacks—Spam

Of course, it is possible to flood a host at the application level. Such an attack may be aimed at exhausting the process table or the available CPU.

Perhaps a disk can be filled by using e-mail or FTP to send a few gigabytes. It's hard to set an absolute upper bound on resource consumption. Apart from the needs of legitimate power users, it's just too easy to send 1 MB a few hundred times instead. Besides, that creates a great deal of receiving processes on your machine, tying it up still further.

The best you can do is provide sufficient resources to handle just about anything (disk space costs are plummeting these days), in the right spots (e.g., separate areas for mail, FTP, and especially precious log data); and make provisions for graceful failure. A mailer that cannot accept and queue an entire incoming mail job should indicate that to the sender. It should not give an "all clear" response until it knows that the message is safely squirreled away.

E-mail spam is now a fact of life. Most Internet users receive a handful of these messages every day, and that is after their service provider may have filtered out the more obvious garbage. The extent of the problem became obvious when we set up an account on one of the free Web-based mail servers and used it to sell one item in an online auction. Although the account was never used for anything else, every time we check it (about once a month), there are hundreds of unsolicited mail messages, touting all sorts of Web sites for losing weight, making money fast, and fulfilling other online fantasies. For most people, spam is a nuisance they've come to accept. However, the kind of spam caused by e-mail viruses and worms (and users who should know better) has brought many a mailer to its knees.

5.8.3 DDoS

DDoS attacks received international attention when they successfully brought down some of the best known Web portals in February, 2000. (Coincidentally, this happened shortly after one of

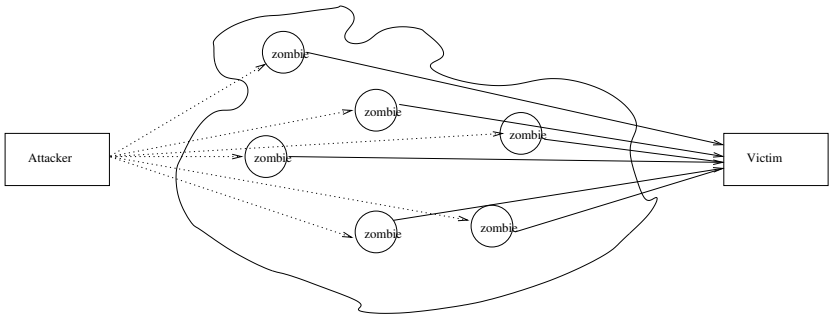

Figure 5.1: Distributed Denial-of-Service Attack The attacker sends a message to the command node. The command node then sends a message to the zombies, which in turn flood the target with traffic.

---

us (Steve) described how these attacks work at *The North American Network Operators' Group (NANOG)*. The *Washington Post* wondered in print if there was a connection. We doubt it, but don't know for sure.) DDoS attacks are DOS attacks that come simultaneously from many hosts conscripted from all over the net. They work as follows (also see Figure 5.1):

1. The attacker uses common exploits to install a *zombie* program on as many machines as he can, all over the Internet, in many different administrative domains. The zombie binds to a port and waits for instructions.

2. The attacker installs a *command* program somewhere on the Internet. The command node has a list of all of the locations of the zombies. The command node then waits for instructions.

3. The attacker waits.

4. When it is time to strike, the attacker sends a message to the command node indicating the address of the target. The command node then sends a message to each of the zombies with the address of the target.

5. At once, the zombies flood the target with enough traffic to overwhelm it.

The message from command node to zombie usually has a spoofed source address, and can even use cryptography to make the messages harder to identify. The traffic from the zombies can be sent with spoofed IP source addresses to make it difficult to trace the actual source, though most attackers don't seem to bother. In addition, the communication from the command node often uses ICMP echo reply, which is allowed by many firewalls.

Several popular DDoS tools, with many variants, are available on the Internet. One of the first was *Tribe Flood Network (TFN)*. It is available in source code form from many sites. The attacker can choose from several flooding techniques, such as UDP flood, TCP SYN flood, ICMP echo request flood, or a smurf attack. A code in the ICMP echo reply from the command node tells the zombies which flood to employ. Other DDoS tools are TFN2K (a more advanced version of

TFN that includes Windows NT and many UNIX flavors), Trinoo, and Stacheldraht. The latter is quite advanced, complete with encrypted connections and an auto-update feature. Imagine a hacker PKI, a web of mistrust?

Newer tools are even more sophisticated. Slapper, a Linux-targeted worm, sets up a peer-to-peer network among the many zombie nodes, which eases the command node's communications problems. Others use IRC channels as their control path.

5.8.4 What to Do About a Denial-of-Service Attack

Denial-of-service attacks are difficult to deal with. We can mitigate an attack, but there are no absolute solutions.

> *Any public service can be abused by the public.*

When you are under one of these attacks, there are four general things you can do about it:

1. Find a way to filter out the bad packets,
2. Improve the processing of the incoming data somehow,
3. Hunt down and shut down the attacking sites, and
4. Add hardware and network capacity to handle your normal load plus the attack.

None of these responses is perfect. You quickly enter an arms race with the attackers, and your success against the attack depends on how far your opponent is willing to go. Let's look at these approaches.

Filter Out the Bad Packets

There may be something specific you can identify in the attacking packets that makes it easy to filter these out without much trouble. Perhaps the packets come from a particular port. They might appear to come from a network that would never support one of your legitimate users. These idiosyncrasies can be quite technical—in one attack, the packets always started with a particular TCP sequence number. You may find yourself deep in the details of TCP and IP when trying to discard evil packets (but see RFC 3514).

The filter may be installed in a router, or even in the kernel of the host under attack. The filter doesn't have to be perfect, and it may be okay to turn away some percentage of your legitimate traffic. The details depend very specifically on the attack and your business. It may be much better to let 80% of your users come in than 0%. It's not ideal, but we didn't promise a perfect solution to these attacks.

Early in the Panix attack, the TCP sequence number was nonrandom, making it easy to filter out the bad packets. The attackers changed this to a random number, and the arms race was on. The return address and now-random sequence number in the attacking packets was generated by the *rand* and *random* functions. Could the pseudorandom sequence be predicted and attacking packets identified? Gene Spafford found that it could, if the attacking host did not use a strong random number generator. One version of the published attack program sent packets with an

Resilience of the Internet—Experts to the Rescue

The Internet was designed to be robust from attack: the packets flow around the outage. We are told that Iraq's packet-switched network was the only one that stayed up during heavy bombing in 1991.

Farmers know that it is dangerous to plant a large area (like Kansas) with the identical strain of wheat. This is called a *monoculture*, and monocultures are prone to common-mode attacks.

The Internet is nearly a monoculture. A host must run some implementation of TCP/IP to participate. Most Internet hosts run the same version of the same software. When a bug is discovered, it will probably be available on millions of hosts. This is a basic advantage that the hackers have, because it is unfeasible and silly for each of us to write our own operating system or TCP/IP implementation.

But it also means that many experts are familiar with the same Internet, and are often quickly available when a new threat arises. They can and do pool their expertise to deal with new and interesting problems. Two examples come to mind, though there have been many others.

When the Morris Worm appeared in 1988, it quickly brought many major sites to their knees. Immediately, several groups disassembled the worm's code, analyzed it, and published their results. Workarounds and vaccines were quickly available, and the worm was pretty much tamed within a week.

When Panix was attacked with the SYN packet denial-of-service attack, a group of TCP/IP implementors quickly formed a closed mailing list and started discussing numerous options for dealing with this problem. Sample code appeared quickly, was criticized and improved, and patches were available from many vendors within a week or two.

The Internet citizen benefits from this sort of cooperation. We cannot always anticipate new threats, but we have many people ready to respond and provide solutions. It is usually easy to install new software, much easier than replanting Kansas.

Of course, if the problem is in hardware ...

unusually low initial TTL field. We could ignore packets with a low TTL value, as nearly all IP implementations use a fairly high initial value. These are the games one has to play at this stage, while the attackers are debugging their packet generator. (Note also that low TTL values can result from *traceroutes*. Do you want to block those?)

There may be other anomalies. Normal packets have certain characteristics that random ones lack. Some commercial products look for these anomalies and use them to drop attack packets.

Typical attack packets have random return IP addresses. If they were a single address or simple range of addresses, we might be able to simply ignore them, unless they appeared to come from an important customer. Given random return addresses, we could try to filter out a few of them on some reasonable basis.

For example, though much of the Internet address space has been allocated, not nearly as much is in use and accessible from the general Internet. Though a company may have an entire /8 network assigned to it, it may only announce a tiny bit externally. We could throw away any random packets that appear to come from the rest of that network.

It would not be hard to construct a bitmap or a Bloom filter [Bloom, 1970] of the 2^{24} addresses that are unassigned or unannounced. Turn off all the multicast nets. Clear any nets that don't appear in the BGP4 list of announced networks. One could even randomly *ping* some of the incoming flow of packets and reject further packets from a net that is unresponsive. Be careful though: Setting the wrong bit in this table could be a fine denial-of-service attack in itself.

Of course, such a bitmap could be quite useful network-wide, and might be a good service for someone to provide. We don't suggest that an actual filter necessarily be implemented with a single bitmap: There are better ways to implement this check that use much less memory. The global routing table keeps hitting size limits, requiring router upgrades.

We might also create a filter that identifies our regular users. When an attack starts, we scan logs for the past month or so to collect the network addresses of our regular users and the ports they use. A filter can check to see if the packet appears to come from a friend, and reject it if it doesn't.

The success of this filter depends on the kind of services we are supplying. It would work better for *telnet* sessions from our typical users than from Web sessions from the general public. E-mail might be filtered well this way: We would still receive mail from our recent correspondents, but unfortunately might turn away new ones. Again, the filter is not perfect, but at least we can transact some business.

In a free society, *shunning* can be a powerful tool to discipline misbehavers. We can decide not to talk to someone, period. Various religious groups like the Amish have used this to enforce their rules. The filters we've discussed can be used to deny access to our services to someone we don't like.

For example, if denial-of-service packets consistently come from a particular university, we can simply cut off the entire university's access to us. This happened to MIT a few years ago; so many hackers were using their hosts that many sites refused to accept packets from the university.

The legitimate users at MIT were having noticeable trouble reaching many sites. The offending department changed their access rules as a result, and most hackers moved on.

Sometimes, the proper defense is legal. There have been a few cases (e.g., *CompuServe v. Cyber Promotions, Inc.*, 962 F.Supp. 1015 (S.D. Ohio 1997)) in which a court has barred a spammer from annoying an ISP's subscribers. We applaud such decisions.

Improve the Processing Software

If you have the source code to your system, you may be able to improve it. This solution is not practical for most sites, which simply lack the time, expertise, and interest in modifying a kernel to cope with a denial-of-service attack. The relevant source is often not available, as in the case of routers or Microsoft products. Such sites ask for help from the vendors, or seek other solutions.

Hunt Them Down Like Dogs

These packets have to come from somewhere. Perhaps we can hunt them down to the source and quench the attack. We don't hold out much hope of actually catching the attacker, as the packet-generating host has almost certainly been subverted by a distant attacker, but maybe we'll get lucky.

The TTL field in the packets may give us a clue to the number of hops between the attacker and us. A typical IP path may hit 20 hops or more, so we have a fair distance to go. But different operating systems have characteristic starting values; this lets us narrow the range considerably.

The return address is probably not going to be helpful. If it is predictable, it is probably easier to simply filter out the packets and ignore them. If the source address is accurate, it should be easy to contact the source and do something about the packet flow, or complain to an intervening ISP. Of course, in a DDoS attack, there may be too many different sources for this to be feasible.

If the return addresses are random and spoofed, we have to trace the packets back through the busy Internet backbones to the source host [Savage *et al.*, 2000]. This requires the understanding and cooperation of the Internet Service Providers. Many ISPs are improving their capabilities to do this.

Will the ISPs cooperate? Most do, when served with a court order. But international boundaries make that tougher.

Is it legal for them to perform the traceback? Is this a wiretap? Do I have a right to see a packet destined for me before it reaches my network?

Perhaps the obvious approach for the ISP is to use router commands to announce the passage of certain packets. Cisco routers have an `IP DEBUG` command that can match and print packets that match a particular pattern. This can be used on each of their routers until the packets are traced back to one of their customers, or another ISP. We are told that this command will hang the router if it is very busy. This has to be repeated for previous hops, probably on different ISPs, perhaps in different countries.

Some routers have other facilities that will help. Cisco's NetFlow, for example, can indicate the interface from which traffic is arriving.

[Stone, 2000] describes an overlay network that can simplify an ISP's traceback problems, but it demands advance planning by the ISP.

If the packets are coming from one of the ISP's own customers, they may contact the customer for further help, or install a filter to prevent this spoofing from that customer. Such a filter is actually a very good idea, and some ISPs have installed them on the routers to their customers. It ensures that the packets coming from a customer have a return address that matches the nets announced for that customer.

Such a filter may slow the router a bit, but the connections to a customer are usually over relatively slow links, like DS1 lines. A typical router can filter at these speeds with plenty of CPU power to spare. More troubling is the extra administrative effort required. When an ISP announces a new net, it will have to change the filter rules in an edge router as well. This does take extra effort, and is another opportunity to make a mistake.

By the way, this filter should not just *drop* spoofed packets—this is useful information that should not be thrown away. *Log* the rejected packets somewhere, and inform the customer that he or she is generating suspicious packets. This alert action can help catch hackers and prevent the misuse of a customer's hosts. It also demonstrates a competence that a competing ISP may not have.

It would be nice to have the Internet's core routers perform similar filtering, rejecting packets with incorrect return addresses. They should already have the appropriate information (from the BGP4 routing tables), and the lookup could be performed in parallel with the destination routing computation. The problem is that many routing paths are asymmetric. This would add cost and complexity to routers, which are already large and expensive. Router vendors and ISPs don't seem to have an incentive to add this filtering.

There are other ways of detecting the source of a packet flow. An ISP can disconnect a major feed for a few seconds and see if the packet flow stops at the target. This simple and alarming technique can be used quickly if you have physical access to the cables. Most clients won't notice the brief outage. Simply disconnect network links until the right one is found.

This can also be done from afar with router commands of various kinds. It has even been suggested that a more cooperative ISP could announce a route to the attacked network, short-circuiting the packets away from a less "clueful" carrier. If this mechanism isn't implemented correctly, it too can be the source of denial-of-service attacks.

One could imagine a command to a router: "Don't forward packets to my net for the next second." We could note the interruption of the incoming packet stream and trace the packets back. This command itself could be used to launch a denial-of-service attack. The command might require a proper cryptographic signature, or perhaps the router only accepts one of these commands every few minutes. There are games one can play with router configurations and routing protocols to do this very quickly, but only the ISP's operations staff can trigger it[3].

A promising approach to congestion control is *Pushback* [Mahajan *et al.*, 2002; Ioannidis and Bellovin, 2002]. The idea is for routers to identify aggregates of traffic that are responsible for congestion. The aggregate traffic is then dropped. Finally, requests to preferentially drop the aggregate traffic are propagated back toward the source of the traffic. The idea is to enhance the service to well-behaved flows that may be sharing links with the bad traffic.

3. See `http://www.nanog.org/mtg-0210/ispsecure.html`, especially pp. 68–76.

Increase the Capacity of the Target

This is probably the most effective remedy for denial-of-service attacks. It can also be the most expensive. If they are flooding our network, we can install a bigger pipe. A faster CPU with more memory may be able to handle the processing. In the Panix attack, a proposal was advanced to change the TCP protocol to require less state for a half-open connection, or to work differently within the current TCP rules.

It's usually hard to increase the capacity of a network link quickly, and expensive as well. It is also disheartening to have to spend that kind of money simply to deal with an attack.

It may be easiest to improve the server's capacity. Commercial operating systems and network server software vary considerably in their efficiency. A smarter software choice may help. We don't advocate particular vendors, but would like to note that the implementations with longer histories tend to be more robust and efficient. They represent the accumulation of more experience.

But the problem won't go away. Some day in the future, after all the network links are encrypted, all the keys are distributed, all the servers are bug-free, all the hosts are secure, and all the users properly authenticated, denial-of-service attacks will still be possible. Well-prepared dissidents will orchestrate well-publicized attacks on popular targets, like governments, major companies, and unpopular individuals. We expect these attacks to be a fact of life on the Internet.

5.8.5 Backscatter

An IP packet has to have a source address—the field is not optional. DOS attackers don't wish to use their own address or a stereotyped address because it may reveal the source of the attack, or at least make the attack packets easy to identify and filter out. Often, they use random return addresses. This makes it easier to measure the attack rate for the Internet as a whole.

When a host is attacked with DOS packets, it does manage to handle some of the load. It responds to the spoofed IP addresses, which means it is spraying return packets across the Internet address space. These packets can be caught with a *packet telescope*, a program that monitors incoming traffic on an announced but unused network.

We actually encountered this effect in 1995, when we announced the then unused AT&T net 12.0.0.0/8 and monitored the incoming packet stream. We caught between 5 and 20 MB per day of random packets from the Internet. Some packets leaked out from networks that were using net 12 internally. Others came from configuration errors of various sorts. But the most interesting packets came from hosts under various IP spoofing attacks. The Bad Guys had chosen AT&T's unused network as a source for their spoofed packets, perhaps as a joke or nod to "the telephone company." What we were seeing were the death cries of hosts all over the net.

In [Moore *et al.*, 2001] this was taken much further. They monitored and analyzed this backscatter traffic to gain an idea of the actual global rate and targets for these attacks. It is rare that we have a technique that gives us an indication of the prevalence of an attack on a global basis. Aside from research uses, this data has commercial value: Many companies monitor clients for trouble, and a general packet telescope is a fine sensor for detecting DOS attacks early.

We used a /8 network to let us catch 1/256 of the randomly addressed packets on the network. Much smaller networks, i.e., smaller telescopes, can still get a good sampling of this

traffic—a /16 network is certainly large enough. By one computation, a /28 (16 hosts) was receiving six or so of these packets per day.

Of course, there's an arms race implied with these techniques. The attackers may want to avoid using return addresses of monitored networks. But if packet telescopes are slipped into various random smaller networks, it may be hard to avoid tipping off the network astronomers.

5.9 Botnets

The zombies used for DDoS attacks are just the tip of the iceberg. Many hackers have constructed *botnets*: groups of *bots*—robots, zombies, and so on—that they can use for a variety of nefarious purposes.

The most obvious, of course, is the DDoS attacks described earlier. But they also use them for distributed vulnerability scanning. After all, why use your own machine for such things when you can use hundreds of other people's machines? Marcus Leech has speculated on using worms for password-cracking or distributed cryptanalysis [Leech, 2002], in an Internet implementation of Quisquater and Desmedt's *Chinese Lottery* [Quisquater and Desmedt, 1991]. Who knows if that's already happening?

The bots are created by traditional means: Trojan horses and especially worms. Ironically, one of the favorite Trojan horses is a booby-trapped bot-builder: The person who runs it thinks that he's building his own botnet, but in fact his bots (and his own machine) have become part of someone else's net.

Using worms to build a botnet—*slapper* is just one example[4]—can be quite devastating, because of the potential for exponential spread [Staniford *et al.*, 2002]. Some worms even look for previously installed back doors, and take over someone else's bots.

The "command node" and the bots communicate in a variety of ways. One of the favorites is IRC: It's already adapted to mass communication, so there's no need for a custom communication infrastructure. The commands are, of course, encrypted. Among the commands are some to cause the bot to update itself with new code—one wouldn't want an out-of-date bot, after all.

5.10 Active Attacks

In the cryptographic literature, there are two types of attacker. The first is a passive adversary, who can eavesdrop on all network communication, with the goal learning as much confidential information as possible. The other is an active intruder, who can modify messages at will, introduce packets into the message stream, or delete messages. Many theoretical papers model a system as a star network, with an attacker in the middle. Every message (packet) goes to the attacker, who can log it, modify it, duplicate it, drop it, and so on. The attacker can also manufacture messages and send them as though they are coming from anyone else.

The attacker needs to be positioned on the network between the communicating victims so that he or she can see the packets going by. The first public description of an active attack against TCP

4. See CERT Advisory CA-2002-27, September 14, 2002.

that utilized sequence number guessing was described in 1985 [Morris, 1985]. While these attacks were considered of theoretical interest at that time, there are now tools available that implement the attack automatically. Tools such as *Hunt*, *Juggernaut*, and *IP-Watcher* are used to hijack TCP connections.

Some active attacks require disabling one of the legitimate parties in the communication (often via some denial-of-service attack), and impersonating it to the other party. An active attack against both parties in an existing TCP connection is more difficult, but it has been done [Joncheray, 1995]. The reason it is harder is because both sides of a TCP connection maintain state that changes every time they send or receive a message. These attacks generally are detectable to a network monitor, because many extra acknowledgment and replayed packets exist, but they may go undetected by the user.

Newer attack tools use ARP-spoofing to plant the man in the middle. If you see console messages warning of ARP information being overwritten, pay attention...

Cryptography at the high layers can be used to resist active attacks at the transport layer, but the only response at that point is to tear down the connection. Link- or network-layer cryptography, such as IPsec, can prevent hijacking attacks. Of course, there can be active attacks at the application level as well. The man-in-the-middle attack against the Diffie-Hellman key agreement protocol is an example of this. (Active attacks at the political layer are outside the scope of this book.)