9

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.

9.1 Stealing Passwords

The easiest way into a computer is usually 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. 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 file name. (A system call did return the file name.)

Some early login programs would service an indefinite number of invalid login attempts. Password-guessing programs were limited by the modem speeds available (which were very low by today's standards). Nothing was logged. A human would try to guess the password based on personal information and the laziness of the password creator.

The battle continued. In UNIX the password file was encrypted [Morris and Thompson, 1979], but world-readable. This led to off-line dictionary attacks [Grampp and Morris, 1984; Klein, 1990; Leong and Tham, 1991; Spafford, 1992a]. They took more CPU time, but the cracking could be done thousands of miles from the machine at risk. The shadow password file attempts to get the administrator out of the game, but many systems don't implement it, and some that do, implement it poorly [Gong *et al.*, 1993].

The battle continues still, on another front. Networks add their own demands and their own risks. New services such as FTP require the same authentication but, curiously, they use a separate routine. All the grizzled code in *login* gets poorly replicated in FTP, yielding more holes (see, for example, CERT Advisory CA-88:01, December 1988). Simple or sophisticated wiretapping code can snatch passwords from the first few packets of a new *telnet* session. Many Bad Guys have libraries of ready-to-install system daemons that log passwords in a secret place, then complete the job. One wonders how often they install software that is more up to date than the stuff they are replacing.

Other network services can aim for the password file. Can *tftp* or *ftp* read `/etc/passwd`? Does the password file in the FTP directory contain entries that work on the real system? Can an outside user tell the mail system to send an arbitrary file to an outside address?

40 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. Remember, though, the Bad Guy only has to win once.

9.2 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 the principal worry we address 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: ches@research.att.com
Subject: Visitor

Bill, 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
```

It is worth noting 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. (On the other hand, it does avoid having to send a cleartext password via email. 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 13 are recommended. A word of caution, though: 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.

9.3 Bugs and Backdoors

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. The rest is history.

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.

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.

In the case of the Worm, for example, most of the structural safeguards of the Orange Book [DoD, 1985a] would have done no good at all. At most, a high-rated system would have confined the breach to a single security level. But effectively, the Worm was 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 (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 are enforced by the evaluating agency. But as we all know, even the best engineered systems have bugs.

The Worm is a particularly apt lesson, because it illustrates 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 to this, you need to have 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, among them that you cannot write down an input grammar if you don't *know* what is legal. That is, 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, after all, require it—but there are several possible designs that 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 4.5.5.)

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.

9.4 Authentication Failures

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 (i.e., if a firewall screens out forgeries), but hackers can use the *portmapper* 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 timesharing 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 entirely too easy.

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 do secure encryption.

The X11 protocol has more than its share of vulnerabilities. The normal mode of application authentication is by host address, whereas the desired mode is by user. This failure means that any user on the same machine as a legitimate application can connect to your server. There are more sophisticated modes of authentication, but they are hard to use. The so-called *magic cookie* mode uses a random string shared by the client and server. However, no secure means is provided to get the cookie to both ends, and even if there were, it is passed over the network in the clear. Any eavesdropper could pick it up. The DES authentication mode is reasonably secure, but again, no key distribution mechanism is provided. Finally, X11 can use Secure RPC, but as described later, Secure RPC is not as good as it should be.

9.5 Protocol Failures

In the previous section, we discussed situations where everything was working properly, but trustworthy authentication was not possible. Here we consider the dual: 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 any of the RPC-based protocols.

In the cryptographic world, finding holes in protocols is a popular game. Sometimes, the creators made mistakes, plain and simple. More often, the holes arise because of different assumptions. Proving the correctness of cryptographic exchanges is a difficult business and is the subject of 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, for example, the Secure RPC protocol. Although well intentioned (and far better than the simple address-based alternative), it has numerous serious problems.

The first is key distribution. Hosts that wish to communicate securely must know each other's public keys. (Public and private key cryptography is discussed further in Chapter 13.) But this information is transmitted via NIS, itself an insecure, RPC-based service. If that exchange is compromised, the remainder of the authentication steps will fall like dominoes.

Next, hosts must retrieve their own private keys. Again, NIS and RPC are used, although with somewhat more safety, since the private keys are encrypted. But they are protected only by a password, with all the dangers that entails. Ironically, the existence of the file containing public and private keys has created an additional security vulnerability: a new avenue for password-guessing [Gong *et al.*, 1993].

Finally, a temporary session key is negotiated. But the cryptographic algorithm used for the negotiation has been cryptanalyzed; too short a modulus was used [LaMacchia and Odlyzko, 1991].

9.6 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, of course; apart from its value to a password-guesser, the information can be used for social engineering. ("Hey, Robin—the battery on my hand-held 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. 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.

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. But 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 address space and port number scans, looking for hidden hosts and interesting services. 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.

9.7 Denial-of-Service

Some people like to slash car tires or deface walls. Others like to crash other people's computer systems. Vandalism—wanton destructive behavior, for its own sake—has been with us for millennia (were those cave paintings really Neanderthal graffiti?); the emergence of new technologies has simply provided new venues for its devotees. Computer networks are no exception. Thus, there are individuals who use them only to annoy.

Such behavior takes many forms. The crudest and easiest form is to try to fill up someone's disks, by mailing or using FTP to send a few hundred megabytes. 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 lots of receiving processes on your machine, tying it up still further. The best you can do is to provide sufficient resources to handle just about anything (disk space costs much less than a dollar a megabyte these days), in the right spots (i.e., separate areas for mail, FTP, and especially precious log data), and to 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.

Other forms of computer vandalism are more subtle. 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. (And 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.

Aggressive filtering can do a lot to protect you, but there are no absolute guarantees; it can be very hard to tell the difference between genuine messages, ordinary failures, and enemy action.