

## 2

# An Overview of TCP/IP

In this chapter we present an overview of the TCP/IP protocol suite. Although we realize that this is familiar material to many people who read this book, we suggest that you *not* skip the chapter; our focus here is on security, so we discuss the protocols and areas of possible danger in that light.

A word of caution: a security-minded system administrator often has a completely different view of a network service than a user does. These two parties are often at opposite ends of the security/convenience balance. Our viewpoint is tilted toward one end of this balance.

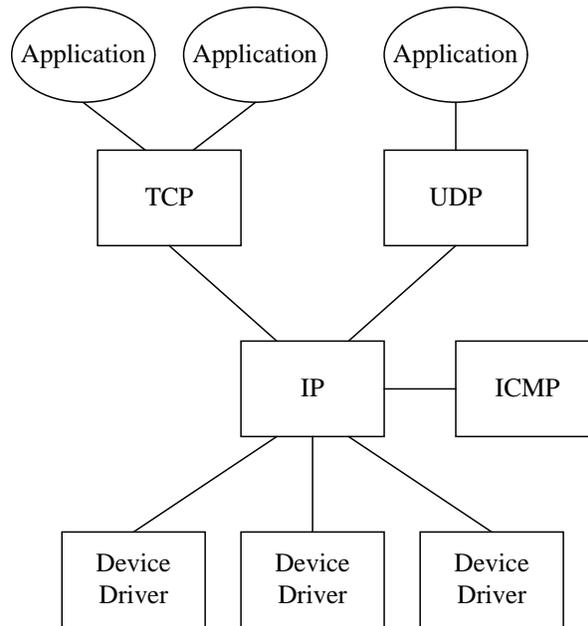
## 2.1 The Different Layers

The phrase *TCP/IP* is the usual shorthand phrase for a collection of communications protocols. It was originally developed under the auspices of the U.S. Defense Advanced Research Projects Agency (then *DARPA*, now *ARPA*), and was deployed on the old ARPANET in 1983. The overview we can present here is necessarily sketchy. For a more thorough picture, the reader is referred to any of a number of books, such as those by Comer [Comer, 1991; Comer and Stevens, 1994] or Stevens [Stevens, 1994].

A schematic of the data flow is shown in Figure 2.1. Each row is a different *protocol layer*. The top layer contains the applications: mail transmission, login, video servers, etc. They call the lower layers to fetch and deliver their data. In the middle of the spider web is the *Internet Protocol (IP)* [Postel, 1981b]. IP is a packet multiplexer. Messages from higher level protocols have an *IP header* prepended to them. They are then sent to the appropriate *device driver* for transmission. We will examine the IP layer first.

### 2.1.1 IP

*IP packets* are the bundles of data that form the foundation for the TCP/IP protocol suite. Every packet carries a 32-bit source and destination address, some option bits, a header checksum, and a payload of data. A typical IP packet is a few hundred bytes long. These packets flow by the billions



**Figure 2.1:** A schematic diagram of the different layers involving TCP/IP.

across the world over Ethernets, serial lines, FDDI rings, packet radio connections, *Asynchronous Transfer Mode (ATM)* links, etc.

There is no notion of a *virtual circuit* or “phone call” at the IP level: every packet stands alone. IP is an unreliable *datagram* service. No guarantees are made that packets will be delivered, delivered only once, or delivered in any particular order. Nor is there any check for packet correctness. The checksum in the IP header covers only that header.

In fact, there is no guarantee that a packet was actually sent from the given source address. In theory, any host can transmit a packet with any source address. Although many operating systems control this field and ensure that it leaves with a correct value, *you cannot rely on the validity of the source address, except under certain carefully controlled circumstances*. Authentication, and security in general, must use mechanisms in higher layers of the protocol.

A packet traveling a long distance will travel through many *hops*. Each hop terminates in a host or *router*, which forwards the packet to the next hop based on routing information. During these travels a packet may be *fragmented* into smaller pieces if it is too long for a hop. A router may drop packets if it is too congested. Packets may arrive out of order, or even duplicated, at the far end. There is usually no notice of these actions: higher protocol layers (i.e., TCP) are supposed to deal with these problems and provide a reliable circuit to the application.

If a packet is too large for the next hop, it is fragmented. That is, it is divided up into two or more packets, each of which has its own IP header, but only a portion of the payload. The fragments make their own separate ways to the ultimate destination. During the trip, fragments

**Table 2.1:** Address Formats

<i>Class</i>	<i>High-order bits</i>	<i>Network Portion</i>	<i>Host Portion</i>	<i>Number of Addresses</i>
A	0	7	24	16,777,214
B	10	14	16	65,534
C	110	21	8	254
D	1110	<i>Multicast group</i>		268,435,456
E	1111	<i>(Experimental use)</i>		–

may be further fragmented. When the pieces arrive at the target machine, they are reassembled. No reassembly is done at intermediate hops.

How IP knows which router to use and how that router determines the proper next hop are questions that are discussed in Section 2.2.

### IP Addresses

Addresses in IP are 32 bits long and are divided into two parts, a *network* portion and a *host* portion. The exact boundary depends on the first few bits of the address. The details are shown in Table 2.1. Host address portions of all 0's and all 1's are reserved.

Generally, the host portion of the address is further divided into a *subnet* and host address. Subnets are used for routing within an organization. The number of bits used for the subnet is determined locally; one very common strategy is to divide a single Class B network into 254 subnetworks.

Most people don't use the actual IP address: they prefer a domain name. The name is usually translated by a special distributed database called the Domain Name System.

These subnet partitioning schemes are wasting addresses and causing the Internet to run out of IP addresses, although there are nowhere near  $2^{32}$  hosts connected to the Internet yet. Proposals are pending to change the interpretation of the IP address formats to expand the address space greatly.

### IP Security Labels

IP has a number of optional fields that may appear, but they are not commonly used. For our purposes, the important ones are the security label and strict and loose source routing. These latter two options are discussed in Section 2.2.

The IP security option [Housley, 1993; Kent, 1991] is currently used primarily by military sites, although there is movement toward defining a commercial variant. Each packet is labeled with the sensitivity of the information it contains. The labels include both a hierarchical component (Secret, Top Secret, etc.) and an optional *category*: nuclear weapons, cryptography, hammer procurement, and so on.

While a complete discussion of security labels and *mandatory access control* is far beyond the scope of this book, a very brief overview is in order. First, the labels indicate the security level of the ultimate sending and receiving processes. A process may not write to a medium with a lower security level, because that would allow the disclosure of confidential information. For obvious reasons, it may not read from a medium containing information more highly classified. The combination of these two restrictions will usually dictate that the processes on either end of a connection be at the exact same level. More information can be found in [Amoroso, 1994].

Some systems, such as UNIX System V/MLS [Flink and Weiss, 1988, 1989], maintain security labels for each process. They can thus attach the appropriate option field to each packet. For more conventional computers, the router can attach the option to all packets received on a given wire.

Within the network itself, the primary purpose of security labels is to constrain routing decisions. A packet marked “Top Secret” may not be transmitted over an insecure link cleared only for “Bottom Secret” traffic. A secondary use is to control cryptographic equipment; that selfsame packet may indeed be routed over an insecure circuit if properly encrypted with an algorithm and key rated for “Top Secret” messages.

### 2.1.2 ARP

IP packets are usually sent over Ethernets. The Ethernet devices do not understand the 32-bit IP addresses: they transmit Ethernet packets with 48-bit Ethernet addresses. Therefore, an IP driver must translate an IP destination address into an Ethernet destination address. While there are some static or algorithmic mappings between these two types of addresses, a table lookup is usually required. The *Address Resolution Protocol (ARP)* [Plummer, 1982] is used to determine these mappings.

ARP works by sending out an Ethernet broadcast packet containing the desired IP address. That destination host, or another system acting on its behalf, replies with a packet containing the IP and Ethernet address pair. This is cached by the sender to reduce unnecessary ARP traffic.

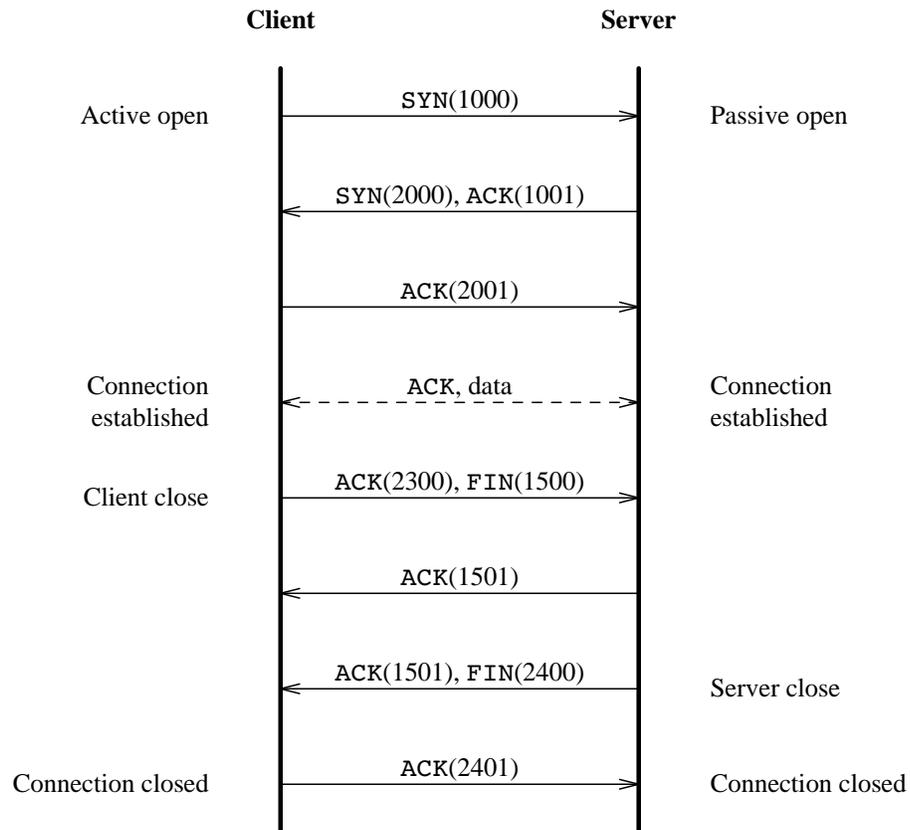
There is some risk here if untrusted nodes have write-access to the local net. Such a machine could emit phony ARP messages and divert all traffic to itself; it could then either impersonate some machines or simply modify the data streams *en passant*.

The ARP mechanism is usually automatic. On special security networks, the ARP mappings may be statically hard-wired, and the automatic protocol suppressed to prevent interference.

### 2.1.3 TCP

The *Transport Control Protocol (TCP)* [Postel, 1981c] provides reliable *virtual circuits* to user processes. Lost or damaged packets are retransmitted; incoming packets are shuffled around, if necessary, to match the original order of transmission.

The ordering is maintained by *sequence numbers* in every packet. Each byte sent, as well as the open and close requests, are numbered individually (Figure 2.2). All packets except for the very first TCP packet sent during a conversation contain an *acknowledgment* number; it gives the sequence number of the last sequential byte successfully received.



**Figure 2.2:** Picture of a sample TCP session. The initial packet, with the SYN (“synchronize,” or open request) bit set, transmits the initial sequence number for its side of the connection. The `initial sequence numbers` are random. All subsequent packets have the ACK (“acknowledge”) bit set. Note the acknowledgment of the FIN (“final”) bit and the independent close operations.

Every TCP message is marked as being from a particular host and *port number*, and to a destination host and port. The 4-tuple

$\langle \text{localhost}, \text{localport}, \text{remotehost}, \text{remoteport} \rangle$

uniquely identifies a particular circuit. It is not only permissible, it is quite common to have many different circuits on a machine with the same local port number; everything will behave properly as long as either the remote address or port number differ.

*Servers*—processes that wish to provide some service via TCP—*listen* on particular ports. By convention server ports are low-numbered. This convention is not always honored, which can cause security problems, as we’ll see later. The port numbers for all of the standard services are assumed to be known to the caller. A listening port is in some sense half-open; only the local host and port number are known. (Strictly speaking, not even the local host address need be known. Computers can have more than one IP address, and connection requests can usually be addressed to any of the legal addresses for that machine.) When a connection request packet arrives, the other fields are filled in. If appropriate, the local operating system will clone the listening connection so that further requests for the same port may be honored as well.

*Clients* use the offered services. Client processes rarely ask for specific port numbers on their local host, although they are allowed to do so. They normally receive whatever port number their local operating system chooses to assign to them.

Most versions of TCP and UDP for UNIX systems enforce the rule that only the superuser (*root*) can create a port numbered less than 1024. These are *privileged ports*. The intent is that remote systems can trust the authenticity of information written to such ports. The restriction is a convention only, and is *not* required by the protocol specification. Conforming implementations need not honor this. In any event, it is meaningless on single-user machines such as PCs. The implications are clear: one can trust the sanctity of the port number only if one is certain that the originating system has such a rule, is capable of enforcing it, and is administered properly.

The sequence numbers mentioned earlier have another function. Because the initial sequence number for new connections changes constantly, it is possible for TCP to detect stale packets from previous incarnations of the same circuit (i.e., from previous uses of the same 4-tuple). There is also a modest security benefit: a connection cannot be fully established until both sides have acknowledged the other’s initial sequence number. This is shown in the third line of Figure 2.2.

2 But there is a threat lurking here. If an attacker can predict the target’s choice of starting points—and Morris showed that this was indeed possible under certain circumstances [Morris, 1985; Bellovin, 1989]—then it is possible for the attacker to trick the target into believing that it is talking to a trusted machine. In that case, protocols that depend on the IP source address for authentication (e.g., the “*r*” commands discussed later) can be exploited to penetrate the target system. This is known as a *sequence number attack*.

Two further points are worth noting. First, Morris’s attack depended in part on being able to create a legitimate connection to the target machine. If those are blocked, perhaps by a firewall, the attack would not succeed. Conversely, a gateway machine that extends too much trust to inside machines may be vulnerable, depending on the exact configuration involved. Second, the concept of a sequence number attack can be generalized. Many protocols other than TCP are vulnerable

[Bellovin, 1989]. In fact, TCP's three-way handshake at connection establishment time provides more protection than do some other protocols.

#### 2.1.4 UDP

The *User Datagram Protocol (UDP)* [Postel, 1980] extends to application programs the same level of service used by IP. Delivery is on a best-effort basis; there is no error correction, retransmission, or lost, duplicated, or re-ordered packet detection. Even error detection is optional with UDP.

To compensate for these disadvantages, there is much less overhead. In particular, there is no connection setup. This makes UDP well suited to query/response applications, where the number of messages exchanged is small compared to the connection setup/teardown costs incurred by TCP.

When UDP is used for large transmissions it tends to behave badly on a network. The protocol itself lacks flow control features, so it can swamp hosts and routers and cause extensive packet loss.

UDP uses the same port number and server conventions as does TCP, but in a separate address space. Similarly, servers usually (but not always) inhabit low-numbered ports. There is no notion of a circuit. All packets destined for a given port number are sent to the same process, regardless of the source address or port number.

 It is much easier to spoof UDP packets than TCP packets, since there are no handshakes or sequence numbers. Extreme caution is therefore indicated when using the source address from any such packet. Applications that care *must* make their own arrangements for authentication.

#### 2.1.5 ICMP

The *Internet Control Message Protocol (ICMP)* [Postel, 1981a] is the low-level mechanism used to influence the behavior of TCP and UDP connections. It can be used to inform hosts of a better route to a destination, to report trouble with a route, or to terminate a connection because of network problems. It also supports the single most important low-level monitoring tool for system and network administrators: the *ping* program [Stevens, 1990].

 Many ICMP messages received on a given host are specific to a particular connection or are triggered by a packet sent by that machine. In such cases, the IP header and the first 64 bits of the transport header are included in the ICMP message. The intent is to limit the scope of any changes dictated by ICMP. Thus, a `Redirect` message or a `Destination Unreachable` message should be connection-specific. Unfortunately, older ICMP implementations do not use this extra information. When such a message arrives, *all* connections between some pair of hosts will be affected. If you receive a `Destination Unreachable` saying that some packet could not reach host FOO.COM, all connections to FOO.COM will be torn down. This is true even if the original message was triggered by a port-specific firewall filter; it is therefore considered polite for firewalls to refrain from generating ICMP messages that might tear down legitimate calls originating from the same machine. We should also note that some parts of the

hacker community are fond of abusing ICMP to tear down connections; programs to exploit this vulnerability have been captured.

 Worse things can be done with `Redirect` messages. As explained in the following section, anyone who can tamper with your knowledge of the proper route to a destination can probably penetrate your machine. The `Redirect` messages should be obeyed only by hosts, not routers, and only when the message comes from a router on a directly attached network. However, not all routers (or, in some cases, their administrators) are that careful; it is sometimes possible to abuse ICMP to create new paths to a destination. If that happens, you are in serious trouble indeed.

## 2.2 Routers and Routing Protocols

“Roo’•ting” is what fans do at a football game, what pigs do for truffles under oak trees in the Vaucluse, and what nursery workers intent on propagation do to cuttings from plants. “Rou’•ting” is how one creates a beveled edge on a tabletop or sends a corps of infantrymen into full-scale, disorganized retreat. Either pronunciation is correct for *routing*, which refers to the process of discovering, selecting, and employing paths from one place to another (or to many others) in a network.

*Open Systems Networking: TCP/IP and OSI*  
—DAVID M. PISCITELLO AND A. LYMAN CHAPIN

Routing protocols are mechanisms for the dynamic discovery of the proper paths through the Internet. They are fundamental to the operation of TCP/IP. Routing information establishes two paths: from the calling machine to the destination and back. (The second path is usually the reverse of the first. When they aren’t, it is called an *asymmetric route* and is generally not a good thing.) From a security perspective, it is the return path that is often more important. When a target machine is attacked, what path do the reverse-flowing packets take to the attacking host? If the enemy can somehow subvert the routing mechanisms, then the target can be fooled into believing that the enemy’s machine is really a trusted machine. If that happens, authentication mechanisms that rely on source address verification will fail.

 There are a number of ways to attack the standard routing facilities. The easiest is to employ the IP *loose source route* option. With it, the person initiating a TCP connection can specify an explicit path to the destination, overriding the usual route selection process. According to RFC 1122 [Braden, 1989b], the destination machine must use the inverse of that path as the return route, whether or not it makes any sense, which in turn means that an attacker can impersonate any machine that the target trusts.

The easiest way to defend against source routing problems is to reject packets containing the option. Many routers provide this facility. Source routing is rarely used for legitimate reasons, although those do exist. For example, it can be used for debugging certain network problems. You will do yourself little harm by disabling it. Alternatively, some versions of *rlogind* and *rshd*

will reject connections with source routing present. This option is inferior because there may be other protocols with the same weakness, but without the same protection.

 Another path attackers can take is to play games with the routing protocols themselves. For example, it is relatively easy to inject bogus *Routing Information Protocol (RIP)* [Hedrick, 1988] packets into a network. Hosts and other routers will generally believe them. If the attacking machine is closer to the target than is the real source machine, it is easy to divert traffic. Many implementations of RIP will even accept host-specific routes, which are much harder to detect.

Some routing protocols, such as RIP Version 2 [Malkin, 1993] and *Open Shortest Path First (OSPF)* [Moy, 1991], provide for an authentication field. These are of limited utility for three reasons. First, the only authentication mechanisms currently defined are simple passwords. Anyone who has the ability to play games with routing protocols is also capable of collecting passwords wandering by on the local Ethernet cable. Second, if a legitimate speaker in the routing dialog has been subverted, then its messages—correctly and legitimately signed by the proper source—cannot be trusted. Finally, in most routing protocols each machine speaks only to its neighbors, and they will repeat what they are told, often uncritically. Deception thus spreads.

Not all routing protocols suffer from these defects. Those that involve dialogs between pairs of hosts are harder to subvert, although sequence number attacks, similar to those described earlier, may still succeed. A stronger defense, though, is topological. Routers can and should be configured so that they know what routes can legally appear on a given wire. In general, this can be a difficult matter, but firewall routers are ideally positioned to implement the scheme relatively simply. This notion is discussed further in the following chapter.

## 2.3 The Domain Name System

The *Domain Name System (DNS)* [Mockapetris, 1987a, 1987b; Lottor, 1987; Stahl, 1987] is a distributed database system used to map host names to IP addresses, and vice versa. (Some vendors call DNS *bind*, after a common implementation of it.) In its normal mode of operation, hosts send UDP queries to DNS servers. Servers reply with either the proper answer or with information about smarter servers. Queries can also be made via TCP, but TCP operation is usually reserved for *zone transfers*. Zone transfers are used by backup servers to obtain a full copy of their portion of the name space. They are also used by hackers to obtain a list of targets quickly.

A number of different sorts of records are stored by the DNS. An abbreviated list is shown in Table 2.2.

The DNS name space is tree structured. For ease of operation, subtrees can be delegated to other servers. Two logically distinct trees are used. The first maps host names such as NINET.RESEARCH.ATT.COM to addresses like 192.20.225.3. Other per-host information may optionally be included, such as HINFO or MX records. The second tree is for *inverse queries*, and contains PTR records; in this case, it would map 3.225.20.192.IN-ADDR.ARPA to NINET.RESEARCH.ATT.COM. There is no enforced relationship between the two trees, though some sites have attempted the mandate such a link for some services.

**Table 2.2:** Some DNS record types

<i>Type</i>	<i>Function</i>
A	Address of a particular host.
NS	Name server. Delegates a subtree to another server.
SOA	Start of authority. Denotes start of subtree; contains cache and configuration parameters, and gives the address of the person responsible for the zone.
MX	Mail exchange. Names the host that processes incoming mail for the designated target. The target may contain wild cards such as *.ATT.COM, so that a single MX record can redirect the mail for an entire subtree.
HINFO	Host type and operating system information. Omit it, or supply inaccurate information.
CNAME	An alias for the real name of the host.
PTR	Used to map IP addresses to host names.

**8** This disconnection can lead to trouble. A hacker who controls a portion of the inverse mapping tree can make it lie. That is, the inverse record could falsely contain the name of a machine your machine trusts. The attacker then attempts an *rlogin* to your machine; which, believing the phony record, will accept the call.

Most newer systems are now immune to this attack. After retrieving the putative host name via the DNS, they use that name to obtain its set of IP addresses. If the actual address used for the connection is not in this list, the call is bounced and a security violation logged.

The cross-check can be implemented in either the library subroutine that generates host names from addresses (`gethostbyaddr` on many systems) or in the daemons that are extending trust based on host name. It is important to know how your operating system does the check; if you do not know, you cannot safely replace certain pieces. Regardless, whichever component detects an anomaly should log it.

**9** There is a more damaging variant of this attack. In this version, the attacker contaminates the target's cache of DNS responses prior to initiating the call. When the target does the cross-check, it appears to succeed, and the intruder gains access. A variation on this attack involves flooding the target's DNS server with phony responses, thereby confusing it.

Although the very latest implementations of the DNS software are immune to this, it seems imprudent to assume that there are no more holes. We strongly recommend that exposed machines not rely on name-based authentication. Address-based authentication, though weak, is far better.

There is also a danger in a feature available in many implementations of DNS resolvers [Gavron, 1993]. They allow users to omit trailing levels if the desired name and the user's name have components in common.

For example, suppose someone on FOO.DEPT.BIG.EDU tries to connect to some destination BAR.COM. The resolver would try BAR.COM.DEPT.BIG.EDU, BAR.COM.BIG.EDU, and BAR.COM.EDU before trying the (correct) BAR.COM. Therein lies the risk. If someone were to create a domain

COM.EDU, they could intercept traffic intended for anything under .COM. And if they had any wild card DNS records, the situation would be even worse.

Authentication problems aside, the DNS is problematic for other reasons. It contains a wealth of information about a site: machine names and addresses, organizational structure, etc. Consider, for example, the joy a spy would feel on learning of a machine named FOO.7ESS.ATT.COM, and then being able to dump the entire 7ESS.ATT.COM domain to learn how many computers were allocated to developing this new telephone switch. (As far as we know, there is no 7ESS project within AT&T.)

Keeping this information from the overly curious is hard. Restricting zone transfers to the authorized secondary servers is a good start, but clever attackers can exhaustively search your network address space via DNS inverse queries, giving them a list of host names. From there, they can do forward lookups and retrieve other useful information.

## 2.4 Standard Services

### 2.4.1 SMTP

When the staff of an unnetworked company is asked what benefits of Internet connection they desire, electronic mail heads the list. If you are talking mail transport on the Internet, you are usually talking about the *Simple Mail Transport Protocol (SMTP)* [Postel, 1982; Braden, 1989a].

SMTP transports 7-bit ASCII text characters using a simple, slightly arcane protocol. Here is the log from a sample session. The arrows show the direction of data flow.

```
<--- 220 inet.att.com SMTP
---> HELO A.SOME.EDU
<--- 250 inet.att.com
---> MAIL FROM:<Ferd.Berfle@A.SOME.EDU>
<--- 250 OK
---> RCPT TO:<mark.farkle@research.att.com>
<--- 250 OK
---> DATA
<--- 354 Start mail input; end with <CRLF>.<CRLF>
---> From Ferd.Berfle@A.SOME.EDU Thu Jan 27 21:00:05 EST 1994
---> From: Ferd.Berfle@A.SOME.EDU
---> To: mark.farkle@research.att.com
---> Date: Thu, 27 Jan 94 21:00:05 EST
--->
---> Meet you for lunch after the meeting with Sparkle.
--->
---> Ferd
---> .
--->
<--- 250 OK
.... A.SOME.EDU!Ferd.Berfle sent 273 bytes to research.att.com!mark.farkle
---> QUIT
<--- 221 inet.att.com Terminating
```

Here the remote site, A.SOME.EDU, is transferring mail to the local machine, INET.ATT.COM. It is a simple protocol. Postmasters and hackers learn these commands and occasionally type them by hand.

 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. Then 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* attack, an attack that’s aimed at preventing legitimate use of the machine. Suppose I 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

```
VERFY <postmaster>
VERFY <root>
```

often translate the mail alias to the actual login name. This can give 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 later, 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. 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 in most UNIX software distributions, but you get less than you pay for. *Sendmail* is 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*.

For most 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, 1994]. And even when a mailer’s rewrite rules are relatively easy, as in the

System V Release 4 mailer or AT&T's research *upas* mailer [Presotto, 1985], it can still be difficult to figure out what to do. RFCs 822 and 1123 [Crocker, 1982; Braden, 1989a] give useful advice.

*Sendmail* can be avoided or tamed to some extent, and there are other mailers 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` 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.

12 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 [Borenstein and Freed, 1993] 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="rfc1480.txt";
             site="ds.internic.net";
             access-type="anon-ftp";
             directory="rfc"
```

```
Content-Type: text/plain
```

A MIME-capable mailer would retrieve the RFC for you automatically.

But suppose that a hacker sent a forged message containing this:

```
Content-Type: Message/External-body;
             name=".rhosts";
             site="ftp.visigoth.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?

Other MIME dangers include the ability to mail executable programs and to mail Postscript files that themselves can contain dangerous actions. These problems and others are discussed at some length in the MIME specification.

## 2.4.2 Telnet

*Telnet* provides simple terminal access to a machine. The protocol includes provisions for handling various terminal settings such as raw mode, character echo, etc. 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*.

A *telnet* session can occur between two trusted machines. In this case, a secure *telnet* [Borman, 1993b; Safford *et al.*, 1993a] can be used to encrypt the entire session, protecting the password and session contents.

13 Most *telnet* sessions come from untrusted machines. Neither the calling program, 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 log the entire session. This is a common hacking trick, and we have seen it employed often; see, for example, the shenanigans reported at Texas A&M University [Safford *et al.*, 1993b].

Traditional passwords are not reliable when any part of the communications link is tapped. Hackers are doing this a lot, and they are focusing on major network backbones.<sup>1</sup> *We strongly recommend the use of a one-time password scheme.* The most common are based on some sort of hand-held authenticator (Chapter 5). We defer further discussion of our particular implementation until Chapter 4, except to note one point: our *telnet* server does not call *login* to validate the session. There are no opportunities to play games with *login*. All that is available externally is our much simpler authentication server. The simpler program is easier to verify.

The authenticators can secure a login nicely, but they do not protect the rest of a session. For example, wiretappers can read any proprietary information contained in mail read during the session. If the *telnet* command has been tampered with, it could insert unwanted commands into your session, or it could retain the connection after you think you have logged off. (The same could be done by an opponent who plays games with the wires, but those tricks are very much harder, and—at this point—are not likely unless your opponents are extremely sophisticated.)

It is possible to encrypt *telnet* sessions, as will be discussed in Chapter 13. 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.

### 2.4.3 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 believes in 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 allow NTP servers to discard erroneous inputs; this provides a high degree of protection against deliberate subversion as well.

Knowing the correct time allows you to match log files from different machines. The time-keeping 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

---

<sup>1</sup>See CERT Advisory CA:94:01, February 3, 1994.

```
$ finger smb@research.att.com
[research.att.com]

If you want to send mail to someone at AT&T,
address your mail using the following format:
firstname.lastname@att.com
```

**Figure 2.3:** Output from our *finger* command.

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 created by the NTP daemon can also provide clues to actual penetrations. Hackers are fond of replacing various system commands and of 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.

#### 2.4.4 Looking up People

Two standard protocols, *finger* [Harrenstien, 1977] and *whois* [Harrenstien and White, 1982], are commonly used to look up information about individuals. The former can be quite dangerous.

 The *finger* protocol can be used to get information about either an individual user or the users logged on to a system. An example is shown in Figure 7.1 on page 134. 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-guessers, when the account was last used (seldom or never used accounts are much more likely to have bad

passwords), where the user last connected from (and hence a likely target for an indirect attack), and more.

To be sure, the most important output from *finger*—the mapping between a human-readable name and an electronic mail address—is very important. For this reason, many sites are reluctant to disable *finger*. The point may be moot. If firewalls are used, the gateway machine will not have logins—and hence *finger* data—for most users. Nor, of course, will there be any point to trying a password-guessing attack against the firewall machine.

A reasonable compromise is to install a custom *finger* daemon that consults a sanitized database or that simply tells how to send mail to someone within the organization. The output from our replacement for *fingerd* is shown in Figure 2.3. That said, we still log all requests, in case of other trouble from some site.

The *whois* protocol is much more benign, since it only supplies contact information. The standard Internet-wide servers, at NIC.DDN.MIL and RS.INTERNIC.NET, are limited in their scope. Some organizations run their own, but that is not very common.

## 2.5 RPC-based Protocols

### 2.5.1 RPC and the Portmapper

Sun's *Remote Procedure Call (RPC)* protocol [Sun Microsystems, 1988, 1990] underlies many of the newer services. Unfortunately, many of these services represent potential security problems. 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 ordinary-seeming 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, etc. However, record boundaries are inserted in the TCP-based version.

RPC messages begin with their own header. Included in it are 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. It 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 NFS, ignore it in favor of the IP address), and neither the user-id nor the group-id are worth anything unless the message is from a privileged port. 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 cryptographic authentication using DES, the Data Encryption Standard [NBS, 1977]. This is sometimes called *Secure RPC*. All calls are authenticated using a shared *session key* (see Chapter 13). The session keys are distributed using Diffie-Hellman exponential key exchange (see [Diffie and Hellman, 1976] or Chapter 13), though Sun's version is not strong enough [LaMacchia and Odlyzko, 1991] to resist a sophisticated attacker.

Unfortunately, DES-authenticated RPC is not well integrated into most systems. NFS is the only standard protocol that uses it, though one group has added it to their versions of *telnet* and FTP [Safford *et al.*, 1993a], and some X11 implementations can use it. Furthermore, the key distribution mechanisms are very awkward, and do not scale well for use outside of local area networks.

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]. Unlike ASN.1 [ISO, 1987a, 1987b], 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 the *portmapper*. (Those servers that need privileged ports pick and register unassigned low-numbered ones.) The *portmapper*—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 the *portmapper* 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.

The *portmapper* has other abilities that are even less benign. For example, there is a call to unregister a service, fine fodder for denial-of-service attacks since it is not well authenticated. The *portmapper* is also happy to tell anyone on the network what services you are running (Figure 2.4); 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.)

 The most serious problem with the *portmapper*, though, is its ability to issue indirect calls. To avoid the overhead of the extra roundtrip necessary to determine the real port number, a client can ask that the *portmapper* forward the RPC call to the actual server. But the forwarded message, of necessity, carries the *portmapper*'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 the *portmapper* will do their own filtering. If yours will not, make sure that no outsiders can talk to it. But remember that blocking access to the *portmapper* will not block direct access to the services themselves; it's very easy for an attacker to scan the port number space directly.

```

program vers proto  port
100000    2  tcp    111  portmapper
100000    2  udp    111  portmapper
100029    1  udp    656  keyerv
100026    1  udp    729  bootparam
100021    1  tcp    735  nlockmgr
100021    1  udp   1029  nlockmgr
100021    3  tcp    739  nlockmgr
100021    3  udp   1030  nlockmgr
100020    2  udp   1031  llockmgr
100020    2  tcp    744  llockmgr
100021    2  tcp    747  nlockmgr
100021    2  udp   1032  nlockmgr
100024    1  udp    733  status
100024    1  tcp    736  status
100011    1  udp   3739  rquotad
100001    2  udp   3740  rstatd
100001    3  udp   3740  rstatd
100001    4  udp   3740  rstatd
100002    1  udp   3741  rusersd
100002    2  udp   3741  rusersd
100012    1  udp   3742  sprayd
100008    1  udp   3743  walld
100068    2  udp   3744

```

**Figure 2.4:** A *portmapper* dump. It shows the services that are being run, the version number, and the port number on which they live. Note that many of the port numbers are greater than 1024.

Even without *portmapper*-induced problems, the 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. It is worth a great deal of effort to block RPC calls from the outside.

## 2.5.2 NIS

One of the most dangerous RPC applications 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 so-called *shadow password file*. Such a file holds the actual hashed passwords. They are not visible to anyone who obtains `/etc/passwd` via NIS. But such systems need some mechanism for applications to use when validating passwords. This is done via an RPC-based service, and this service does not log high rates of queries, as might be generated by an attacker.

19 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, etc.

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.

### 2.5.3 NFS

The *Network File System (NFS)* [Sun Microsystems, 1989, 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 we 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. The server's mount daemon, an RPC-based service, checks the client's host name and requested file system against an administrator-supplied list, and verifies the mode of operation (read-only versus read/write). If all is well, the file handle for the root directory of the file system is passed back to the client.

20 Note carefully the implications of this. Any client who retains a root file handle has permanent access to that file system. While standard client software renegotiates access at each mount time, which is typically at reboot time, there is no enforceable requirement that it do so. (Actually, the kernel could have its own access control list. In the name of efficiency, this is not done by typical implementations.) Thus, NFS's mount-based access controls are quite inadequate. It is not possible to change access policies and lock out existing but now-untrusted clients, nor is there any way to guard against users who pass around root file handles. (We know someone who has a collection of them posted on his office wall.)

File handles are normally assigned at file system creation time, via a pseudo-random number generator. (Some older versions of NFS used an insufficiently random—and hence guessable—seed for this process. Reports indicate that successful guessing attacks have indeed taken place.) New handles can only be written 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 is not, in fact, used by the system for anything important.

To the extent that it is available, NFS can use Secure RPC. This guards against address spoofing and replay attacks. But Secure RPC is not available on all platforms and is difficult to deploy under certain circumstances. For example, some machines do not support key change operations unless NIS is in use.

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 the *portmapper* providing addressing information.

NFS poses risks to client machines as well. Someone with privileged access to the server machine 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. The best defense here is 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.

Version 3 is starting to be deployed. Its most notable attribute (for our purposes) is support for transport over TCP. That will make authentication much easier.

## 2.5.4 Andrew

The *Andrew File System (AFS)* [Howard, 1988; Kazar, 1988] is another networked 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 allows files to live on any server within the network, with caching occurring transparently, and as needed.

AFS uses the Kerberos authentication system [Bryant, 1988; Kohl and Neuman, 1993; Miller *et al.*, 1987; Steiner *et al.*, 1988], which is described further in Chapter 13, and a Kerberos-based user identifier mapping scheme. It thus provides a considerably higher degree of safety than does NFS. Furthermore, Kerberos scales better than does secure RPC. That notwithstanding, there have been security problems with some earlier versions of AFS, but those have now been corrected; see, for example, [Honeyman *et al.*, 1992].

## 2.6 File Transfer Protocols

### 2.6.1 TFTP

The *Trivial File Transport Protocol (TFTP)* is a simple UDP-based file transfer mechanism. It has no authentication in the protocol. It is often used to boot 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.

Some routers 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?)

### 2.6.2 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 (Figure 2.5), the user's `ftp` command opens a control channel to the target machine. The lines starting with `---` show the commands that are actually sent over the wire; responses are preceded by a 3-digit code.

```
$ 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 2.5:** A sample FTP session.

Sometimes, such as after a `USER` command is sent, the response code indicates that the daemon has entered some special state where only certain commands are accepted.

The actual data, be it a file transfer or the listing from a directory command, is sent over a separate *data channel*. The server uses port 20 for its end. By default, the client uses the same port number as is used for the control channel. The FTP protocol specification suggests that a single channel be created and kept open for all data transfers during the session. Most common implementations create a new connection for each file. Furthermore, 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. Normally, the client listens on a random port number, and informs the server of this via the `PORT` command. In turn, the server makes a call to the given port.

The protocol does provide a way to have the server pick a new port number and to receive the call instead of initiating it. While 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—we can use this feature for our own ends. See the discussion of the `PASV` command in Chapter 3.

By default, transfers are in ASCII mode. Before sending or receiving a file that has other than printable 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.

*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.

Both FTP and the programs that implement it are real problems 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.
- The protocol uses two TCP connections, complicating the job of gating this service through a firewall. In most cases an outgoing control connection requires an incoming data connection.
- The *ftpd* daemon runs as *root* initially since 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.

On the other hand, anonymous FTP has become a principal standard on the Internet for publishing software, papers, pictures, etc. Most major sites need to have a publicly accessible anonymous FTP repository somewhere. Whether you want it or not, you most likely need it.

There is no doubt that anonymous FTP is a valuable service. It is, after electronic mail, arguably the most important service on the Internet. 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. You can give a hacker no greater gift than a real `/etc/passwd` file. 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. Our is shown in Figure 1.2 on page 12.

Whether or not to create a publicly writable directory for incoming files is quite controversial. While 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 or digital erotica. This repository may be permanent or transitory; in the latter case, individuals desiring anonymity from each other 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. (Note: At all costs, resist the temptation to infect the 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 former politicians.) Our gateway machines clear the incoming file area nightly.

If feasible, 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 of the odd names—can deposit files in there, for your users to retrieve at their leisure.

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` 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. (Note that 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.)

### 2.6.3 FSP—The Sneaky File Transport Protocol

 *FSP*—the name does not stand for anything—is another file transport protocol. It uses a UDP port (often the privileged port 21) to implement a service similar to FTP. It is unofficial, and isn't used very often except by hackers, who have found it easy to install and a convenient tool for shipping their tools and booty around. It does have some uses, though; its primitives are more like NFS's, which makes it more amenable to a decent (i.e., file system-like) user interface. And some of its proponents claim that it's more robust on congested links, though that claim seems dubious: UDP lacks the congestion control of TCP. Still, discovery of FSP traffic is cause for concern, given its history of misuse.

## 2.7 The “r” Commands

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:

- 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 only be permitted from machines where 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 `$HOME/.rhosts` file.
- The caller’s name must correspond to its IP address. (Most current implementations check this. See Section 2.3.)

From a user’s viewpoint, this scheme works fairly well. A user can bless the machines he or she wants to use, and isn’t 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, like *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 lesson is obvious: the permission and ownership structure of the server machine must be set up to prohibit this.

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), attaining *root* access, and leaving 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. There are other system logs that 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 only be made 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 directly accessible from the Internet, and we do not support them to or through our gateways.

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.

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, as is done by the TIS Firewall Toolkit (see Section 4.10). This is safe as long as you do not grant any privileges based on the information thus received.

## 2.8 Information Services

### 2.8.1 World Wide Web

Of late, the growth of what might best be termed *information protocols* has been explosive. These include *gopher* [Anklesaria *et al.*, 1993], *Wide Area Information Servers (WAIS)*, and others, sometimes lumped together under the rubric *World Wide Web (WWW)*. While they differ greatly in detail, there are some essential points of similarity in how they operate.

Generally, a host contacts a server, sends a query or information pointer, and receives a response. The response may either be a file to be displayed or it may be a pointer or set of 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. For example, the *gopher* protocol has a *uuencode* format, which includes a file name and mode. Blindly believing such information is obviously quite dangerous.

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 pointers. These pointers often have file names embedded in them [Berners-Lee, 1993]. While the servers do attempt to verify that the requested files are authorized for transfer, the verification process can be (and, in fact, has been) buggy. Failures here can let outsiders retrieve any file on the server's machine.

We would very much prefer a pointer syntax that included an optional field for a cryptographic checksum of the information. That would make the pointers self-validating, and would prevent outsiders from concocting them out of whole cloth. But the problem is a difficult one.

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, say, a version of *telnet* becomes popular that uses preauthenticated connections, the same stunt could cause someone to log in and execute various commands on behalf of the attacker.

 The greatest dangers in this vein result when the server shares a directory tree with anonymous FTP. In that case, an attacker can first deposit control files and then ask the information server to interpret them. 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.)

If, on the other hand, the server initiates a connection in response to a user's request—and *gopherd* will do that for FTP under certain circumstances—there is a very different problem. The connection, though initiated on behalf of the client, appears to come from the server's IP address. Thus, any tests done by IP address will give the wrong result. In effect, address-spoofing *gopherd* will permit laundering of FTP requests. This can have practical implications, as discussed in Section 4.5.5.

28 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*. But even this may not suffice, because the interpreters themselves must reside in this area. We see no good solutions, other than to urge great care in writing the scripts.

## 2.8.2 NNTP—The 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 on how NNTP should be passed through firewalls.

The obvious way is to treat it the same as mail. That is, incoming and outgoing news article 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, etc. 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.

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. The gateway would use a relay program, similar to that described in Chapter 4, to let the news articles pass directly to the inside news hub.

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 very similar dangers.

### 2.8.3 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 Class D multicast address identify the *multicast group* to which a packet is destined. Hosts may belong to zero or more multicast groups.

Since most commercial routers do not yet support multicasting, some hosts are used as multicast routers to forward the packets. 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* [Deering, 1989]. 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.

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.

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 mid-flight.) 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, since many hosts avoid selecting numbers with the sign bit on. The session directory program provides hooks to 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; see Appendix B for a list.

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.

## 2.9 The X11 System

X11 [Scheifler and Gettys, 1992] is the dominant windowing system used on the Internet today. It uses the network for communication between applications and the I/O devices (the screen, the mouse, etc.), 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.

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, etc. 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, there's another mechanism that will let you poll the keyboard up/down status map, and that one can't be locked out.

 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, etc. 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. Unfortunately, they are not as useful as one might hope.

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 granularity of this scheme is 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.

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 as of this writing it is not widely available. Standardized Kerberos support is scheduled for the next major release of X11.

The best current alternative, if you have it available, is Secure RPC. It provides a key distribution mechanism and a reasonably secure authentication mechanism. But be wary of the problems with Secure RPC in general.

## 2.10 Patterns of Trust

A common thread running through this chapter is that computers often trust each other. This is well and good for machines under common control; indeed, it is often necessary to their usability. But the web of trust often spreads far wider than it should. It is a major part of a security administrator's job to ascertain and control which machines trust which, for what, and by what mechanisms. The address-based mechanisms used by many of the standard protocols are inadequate in high-threat environments such as gateways, and often internally as well.

The purpose of a firewall gateway is to sever the web of trust at certain key points. As we shall see in the next chapter, a gateway machine trusts very few others, and only for certain functions. It may trust everyone for mail, but only one or two for netnews. Anonymous FTP may be supported, but no other type; its trust policies do not permit nonanonymous logins. More precisely, they do not permit logins that have access to other than a limited area of the file system, as mediated by a kernel mechanism (for UNIX systems, that is `chroot`); the *ftpd* server is far too complex to be verified. Similarly, the gateway permits pass-through logins via *telnet*, but only after demanding strong authentication.

One could certainly pick other trust policies. Arguably, incoming *telnet* sessions should not be permitted, since an eavesdropper could spy on mail being read by a traveler or an active attacker could take over a *telnet* session. The details of a policy will differ from place to place. The important thing is to pick a policy explicitly, rather than having one put in place by the actions of myriad vendors and system administrators.