4

How to Build an Application-Level Gateway

In this chapter we build an application-level gateway with a high level of security. The configuration described is actually in use as the gateway serving the research community at AT&T Bell Laboratories. Notice that we do not mind describing this gateway in detail. The security of the gateway does not depend on secrecy or on "security by obscurity." There are no hidden trap doors or secret entrances. In fact, early in the installation the outside machine became so secure we lost the ability to log in to it!

We built the gateway with standard UNIX computers, off-the-shelf routers, some simple software tools, and some publicly available BSD-derived network daemons. We have configured the gateway to supply most of the current functions such a gateway can safely handle. You can build a very similar one.

Building this gateway is not hard, but it does take some time. Lots of assembly is required. Your gateway may well differ from this one in detail. The differences may be based on cost, complexity, and safety/hubris/stance. Any of these variations can erect huge obstacles to the hacker, while giving reasonable service to the authorized user.

If this construction project seems daunting or unnecessary to you, see the last section in this chapter on commercial products. We hope that after you read this, you can evaluate the various commercial products yourself to determine their suitability.

4.1 Policy

Our gateway is constructed based on the following policies and assumptions:

- We cannot trust corporate system administrators to keep their machines secure. We are not even sure that *we* can do it, except perhaps in the case of a minimalist gateway machine. Therefore, we cannot afford to allow IP connectivity between the corporate networks and the Internet.

- We trust internal users to keep our company's secrets to themselves. We are not concerned about our people using *ftp* to send out our secrets. If they want to steal information, there are easier ways for them to do it. As we discussed earlier, levels of security should be commensurate. There's no point to building a very high wall against outgoing *ftp*, while not worrying about pocket-sized tape cassettes that hold several gigabytes of data. Others differ on this point; they prefer to install protection against software being shipped out by an outsider who has penetrated their internal networks.
- We don't trust anyone on the Internet without strong authentication. Passwords are not good enough. They are too easily guessed or stolen.
- We will allow through what services we can. The rest remain unavailable to our users. If there is a very strong business case for more access, a sacrificial host may be placed outside the firewall. The machine's administrators take responsibility for keeping it secure.
- Our gateway only minimizes the considerable threat from the masses on the Internet. There are numerous other security threats to the company.

4.2 Hardware Configuration Options

We have used three general application gateway configurations. They are shown in Figure 4.1. Plan A is single-machine with a port in each world. Our first gateway had this configuration. It is equivalent to the basic schematic shown earlier in Figure 3.1, where the host itself incorporates both filters. Though it is much more secure than a packet filtering router, it has some annoying configuration problems and lacks a fail-safe design. Configuration questions:

1. How does the Domain Name Server provide host address translations for both inside and outside access, without leaking naming information to the outside? This requires careful and tricky DNS configuration.
2. Can IP routing be turned off in the host? Can you be sure it has been turned off? Will ICMP `Redirect`s change this behavior? What about IP source routing? Answers to these questions must come from competent analysis of the kernel's source code, backed up by careful experimentation.
3. This host will offer different services on each interface. What mechanism is used to implement this? On most hosts, the same services are offered on all network interfaces. Standard software tends to lack a mechanism for restricting access in this way.
4. What are the consequences if a hacker subverts the gateway host? How would you detect this event?

Despite these problems, many people (and most commercial products) implement a firewall this way. We believe that our future gateway, which will have to support video rate connections, will have to use a modified Plan A for efficiency reasons.

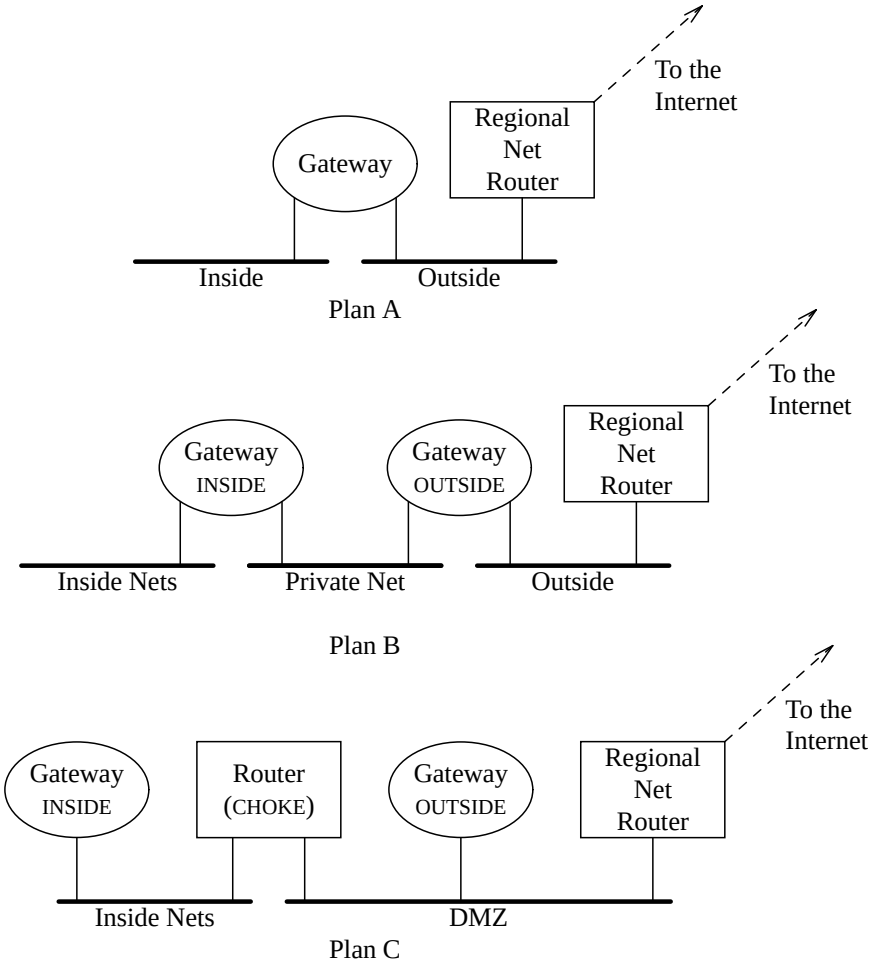

Figure 4.1: Three configurations for an application-level gateway.

Our Many Gateways

AT&T's first gateway to the Arpanet used Plan A. It kept out the Bad Guys, including the Internet Worm. Some luck was involved, though, and the basic design could have allowed the Worm through.

This led to the second gateway, which used Plan B. This worked for a number of years, but became swamped with the load. The major bottleneck was the private network connection, which we implemented with a Datakit VCS. This research interface to the Datakit network was slow (30 KB/sec). and a little flaky under heavy loads.

This led to the recent installation of the third gateway, which uses Plan C. Plan C was selected because the spare hosts we have lying around, MIPS Magnums, only have a single Ethernet connection. Double Ethernet connections let us hide the private link behind the outside machine, which is a bit safer. Our Plan C gateway can transfer data at better than 200 KB/sec, more than enough to saturate our link to the Internet.

Plan B, the "belt-and-suspenders" configuration of Figure 3.11, greatly improves the stance over Plan A. The outside and inside hosts are linked by a private network. The only destination the outside host can reach is the inside one. The inside machine does not trust the outside host. It offers a couple of services to the outside host, typically an authenticated login and a mail gateway. Another service might be a logging drop-safe. The outside machine offers the outside world the usual array of services: external DNS, FTP, SMTP mail, and *telnet* access to the authenticated login.

Plan C is a another popular configuration. It is often used with routers that filter on output only (Figure 3.10). The outside bastion host provides those popular, dirty network services to the world. The choke machine, which can be a router or packet-screening host, limits access to the inside. The inside machine can provide a contact point for inside access. Although the topology differs from Plan B, it implements almost the same division of functions. The choke router enforces a private link between the bastion host and the inside machine.

Plan C offers the outside world two points of attack: the bastion host and the choke machine. In particular, CHOKE must be protected very carefully, since it is connected to both sides. If the router is compromised, the gateway is breached.

In all three plans, the outgoing services may be bridged at the application level as well. In our case, we provide a TCP circuit-level service for our internal users.

The key point to this gateway is that the two machines provide a belt-and-suspenders approach to security. The outside machine takes its best shot at being secure, but the two simple services

on the inside machine ensure that a successful hacker on the outside computer faces an abruptly simple and impenetrable interface on the inner computer.

It is Plan C that we describe here. It is trickier to implement than Plan B is, and provides a number of useful illustrations of the problems and approaches involved. For our example, we will use two MIPS Magnums (plus a spare) with *plenty* of disk space and a two-port Cisco router. (Nothing we did was MIPS-specific; we just happened to have the necessary machines available. We could have built the exact same gateway using virtually any modern UNIX system. The reader may prefer to use the BSDI UNIX system, since the source code is inexpensive and the system has several provisions for gateway use. Linux is another good choice.) The Magnums are running RISC/OS 4.52, which is old but reliable. We don't plan to upgrade the operating system on the Magnums. We will upgrade the firmware on the Cisco router after a respectable soaking period. The leading edge is often the bleeding edge; we are in no rush to upgrade.

4.3 Initial Installation

Before the computers are connected to any networks, we arrange to get easy access to their consoles. These should be through serial lines guarded by passwords and accessible by dialing in from home. We have a switch installed that selects between a local console terminal and the remote access. (Of course, the switch is always in the "local" position when one tries to access the console remotely.) It is important not to connect to these consoles from the outside Internet. If someone taps that session, the outside machine is breached.

The system administration is better done through the console than a network connection, which could be tapped. Physical access to the console is less convenient for system administration, but should be impossible for a typical hacker.

During initial configuration, it makes sense to connect the hosts to a safe internal network. Here are some of the steps to remember:

- For Plans A and B, turn off IP forwarding and source routing in the kernels. Get help from the vendor if you can't figure out how to do this. With Plan C, these are not so important, but you still must ensure that source-routed packets do not pass through CHOKE, and you need to protect OUTSIDE against source-address spoofing.
- The kernel may well need some reconfiguration, because these gateway machines have unusual usage patterns. Some things to check:
  - Is it configured for enough processes? For our arrangement, each TCP connection requires two processes (we don't use `select` for portability reasons). We have 2000 processes configured.
  - Is the per-account process limit high enough, probably nearly the same as the number of processes? The relay programs (see Section 4.4.2) generally all run under the same user name.
  - Are there enough `mbuf`'s configured? These TCP-based solutions are network-intensive.

  - What about other resources, such as memory management tables, open file descriptor tables, etc.?

- Don't skimp on the hardware supplied for each machine. Use plenty of memory, and make sure it is easy to get more. We run with 32 MB on each host. Have plenty of large disk drives. FTP, the logs, and the spool directory all need plenty of space. Remember that disks eventually go bad; have spare partitions and drives ready.

- BSD-based kernels require a lot of swap space if there are a lot of processes, even though we don't plan to swap a single byte to disk.

- Delete *all* network services from both machines. This means the `/etc/inetd.conf` should contain only comment lines. In the startup files, comment out NFS, NIS (Yellow Pages), *sendmail*, and everything else that starts up a network connection.

- Remove the compilers and any other unneeded software from the external host. In particular, remove `setuid` programs that aren't needed or understood.

- Remove any unneeded accounts from both systems. There shouldn't be any accounts other than *root* and the usual system administration accounts. Printer spoolers, administration menus, and the like are not needed. Delete them.

- Reboot these machines and check `netstat -a` to make sure all network services are gone.

- Set `/etc/motd` to warn all users that they might be monitored and prosecuted. On the inside machine, warn *any* user that they are not allowed on the machine. The warning against unauthorized use is probably superfluous, although folklore indicates otherwise. The notice about monitoring is considered necessary by some legal authorities; see Chapter 12.

- Configure the disk partitions. Remember that the outside world has the ability to fill the logs, spool directory, and FTP directories. Each of these should be in a separate large disk partition.

- Connect the external host to its final location on the external network.

- Add the appropriate permanent ARP entry for the router on both hosts. If the router doesn't announce its ARP information, only our two hosts will know how to reach it.

- On the external host, add the appropriate default static routing. Do not run *routed* on the external host: whose information would you trust, anyway? Just set the default route to the regional network's router; they can deal with external routing concerns.

- Take a full dump of both computers and save the tapes *forever*. Make sure they are readable. Do this before plugging in the cable that allows external access for the first time. These are "day 0 backups," and they are your last resort if someone breaks into your machines.

Now it is time to configure CHOKE, the router. This router has an unusual configuration, because only two hosts need to use it. Here are some things to do:

- Turn off *telnet* access to the router. On the Cisco, this involves setting access controls on the virtual terminals. All access should be from the console.
- Turn off ARP and route processing on the router. Everything will be done with static tables. In particular, the router should not respond to ARP requests. Here's a list from our Cisco router:

```
no service finger
no ip redirects
no ip route-cache
no ip proxy-arp
no mop enabled
no ip unreachables
```

  Some of these go in the interface definitions. Others are general to the whole router.
- Install an ARP entry for the external and internal host. No routing tables are needed: the router will talk only to the two locally connected networks.
- Set up an initial packet filter that denies all access to all hosts. One could then permit all access between the inside and outside host. We prefer to open specific connections in the style of a classic packet filter.
- Just for safety, configure the external router on the regional network to block any packets to CHOKE's Ethernet address. If your external router won't let you do this, install a phony static ARP entry instead.

The gateway's skeleton is now ready. We need some simple tools before we install each service.

4.4 Gateway Tools

Here are some simple tools that are vital for constructing a gateway. We will discuss others later.

4.4.1 A TCP Wrapper

In a BSD-based system, a program can create and listen to its own socket, or the daemon *inetd* can listen for calls on behalf of the service and call the appropriate program with the socket connected to the standard input and output of the process. A program must create its own sockets when multiple connections are handled by the same process. But most services use a separate process for each service invocation.

A gateway designer prefers services that can be implemented using *inetd*. It has a simple interface and a single file (`/etc/inetd.conf`) to watch. A very simple `/etc/inetd.conf` file might look like this:

```
#
# Internal procedures
#
echo    stream  tcp     nowait  root    internal
discard stream  tcp     nowait  root    internal
chargen stream  tcp     nowait  root    internal
daytime stream  tcp     nowait  root    internal
time    stream  tcp     nowait  root    internal
echo    dgram   udp     wait    root    internal
discard dgram   udp     wait    root    internal
chargen dgram   udp     wait    root    internal
daytime dgram   udp     wait    root    internal
time    dgram   udp     wait    root    internal
#
# real services
#
smtp    stream  tcp     nowait  uucp    /v/lib/upas/smtpd  smtpd -H
```

In this example, all lines except the last implement standard and probably harmless network services. The last line supports the SMTP service with the *upas* daemon `/v/lib/upas/smtpd`, which runs under account *uucp*. The program need only read and write to standard input and standard output to provide the service. It is easy to implement a simple service with a shell script, and we often do so with our TCP traps and sensors.

Hackers are very interested in seeing `/etc/inetd.conf`: It gives them a list of services to attack. Our pet hacker Berferd (see Chapter 10) tried to obtain this file several times, and was quite interested in it when he was in the Jail. We had to take special care to make sure the Jail's version of this file matched Berferd's concept of our simulated machine.

In the first version of our gateway, we added logging to *inetd* to track all connection attempts:

```
Jun 30 09:19:16 inet inetd[121]: smtp request from 128.174.5.98 pid 2655
Jun 30 09:19:36 inet inetd[121]: exit 2655
```

On our gateway, this log is half a megabyte long each day. We've seldom used it except when there's serious hostile activity going on; then, it's extremely useful to see what services the attacker is investigating.

A better way to implement that sort of logging is to use Wietse Venema's *TCP wrapper* [Venema, 1992]. Apart from logging all connection requests, it provides a convenient hook for access control. To use the wrapper, you install it in `/etc/inetd.conf` as the program to execute for the protected services. When it is done, it `exec`'s the actual server.

A wrapped version of the earlier SMTP sample might be:

```
smtp  stream  tcp  nowait  uucp  /v/gate/tcpd  /v/lib/upas/smtpd
```

The wrapper (`/v/gate/tcpd`) checks `/etc/hosts.allow` to decide whether to accept the connection. The TCP wrapper scans down the file until a match is found. A corresponding verb, such as *accept*, *deny*, *log*, etc., is then applied. If there is no match, the connection would be denied. (We much prefer this approach to the older one, which used two files, `/etc/hosts.allow` and `/etc/hosts.deny`. We'd be even happier with a version that used a file per service. It's much easier in that case to see *all* of the rules that apply to a given service, and to avoid accidental changes to one service when modifying the rules for another.)

Can You Trust the TCP Wrapper?

The TCP wrapper provides for access control on the basis of source IP address. But is that really secure? Elsewhere, we've indicated that we don't trust that as an authentication mechanism.

The answer here is that we don't trust it that much. Primarily, we use it to distinguish between "inside" and "outside" machines. For that, it can be trusted, since the router is configured to prevent address spoofing. Second, we use only numeric IP addresses in our access lists, not host names. That avoids any reliance on the integrity of the name service. Third, on those occasions where we do grant some privilege to outsiders based on their IP address, we either demand additional strong authentication, or we give away comparatively trivial resources.

Even if these features are not trusted, a wrapper is still useful for logging.

It's also worth asking if the wrapper program itself is trustworthy. We're happy with it for now, although it's getting a bit too big and feature-full for our tastes. We may eventually replace it with a simpler program based on the regular expression library.

A quirk of the wrapper's implementation requires that the wrapper be able to determine the caller's host name, even if only addresses are used in the permissions file. This may require a bit of extra effort; OUTSIDE will receive many calls from INSIDE, a machine that isn't accessible from the Internet, and hence may not be listed in the ordinary DNS database. In fact, we have found this lookup to be a needless performance problem.

The TCP wrapper is a fine tool for enforcing gateway connection policy. This is particularly true in a Plan C arrangement, where there will be private connections between the internal and bastion hosts to implement various relays. We don't want outsiders using these services directly.

4.4.2 Relay

Relay is a handy tool for an application-level gateway. It is a small program that copies bytes between two network connections: The computer acts as a wire. For example, imagine that we wish to offer an external machine our printing service: TCP port 515, which is listed in `/etc/services` as `printer`. We wish to relay any connection to this port to the same port on our printer server inside. The `/etc/inetd.conf` entry would be:

```
printer stream tcp nowait daemon /v/gate/relay relay \
        tcp!pserver!printer
```

Relay opens a connection to the printer port on PSERVER, then copies bytes in both directions until the connection closes. Obviously, the CHOKE router would have to be configured to allow this

connection. The TCP wrapper could restrict this print service to particular external machines.

Relay may appear to be a simple program for the C novice, but there are several subtleties to it. First, a TCP connection can "half-close," leaving one direction open and the other closed. The program must be careful to use `shutdown` calls to signal a close, and listen for hang-up signals. The shutdown sequence can be especially important if two *relay*s are running back-to-back in a multimachine gateway. Connections must close properly without leaving spare processes lying around. The choice of using two processes versus a single one using `select` has portability and efficiency concerns. Our version does not pass on the TCP `URGENT` pointer; it probably should. Finally, it might be useful to implement a timeout in some circumstances.

4.4.3 A Better *Telnetd*

The standard *telnetd* command does a good job handling the *telnet* protocol. But it is less flexible than it should be when talking to the host. We've made a number of changes, notably to let it invoke some program other than *login* (see the box on page 97) and to create a shell environment variable identifying the source of the call. (Come to think of it, the TCP wrapper could do this, too.)

4.4.4 Support for Outgoing FTP Access

As we have described, outgoing FTP sessions normally require an incoming TCP call. To support this, our proxy service can listen on a newly created socket. The port number is passed back to the caller, which generates the appropriate FTP `PORT` command. The call is thus outgoing from the user's machine to the firewall, but incoming from the FTP server.

We don't let these sockets hang around long. If the server doesn't establish its connection quickly enough, the socket will be closed and the request aborted.

4.5 Installing Services

We now have the tools to start installing services. These should be installed carefully, one by one, with plenty of testing and paranoia. In the following steps, the external bastion host is called OUTSIDE and the internal host INSIDE. In the actual configuration of the files and router, either the actual domain name or numeric IP address is needed. We leave it to the reader to supply the appropriate values. Note that if you supply a name, and the name translation mechanism is subverted, then you can lose everything. In our gateway, we use the less convenient but safer numeric IP addresses. A reasonable compromise might be to build the files using machine names, and to have a trusted preprocessor do the translations for us.

4.5.1 Mail Delivery

In our gateway, the bastion host receives and sends all mail to external hosts. The mailer is configured to handle `MX` entries, incorrect headers, and other mail problems. All incoming and

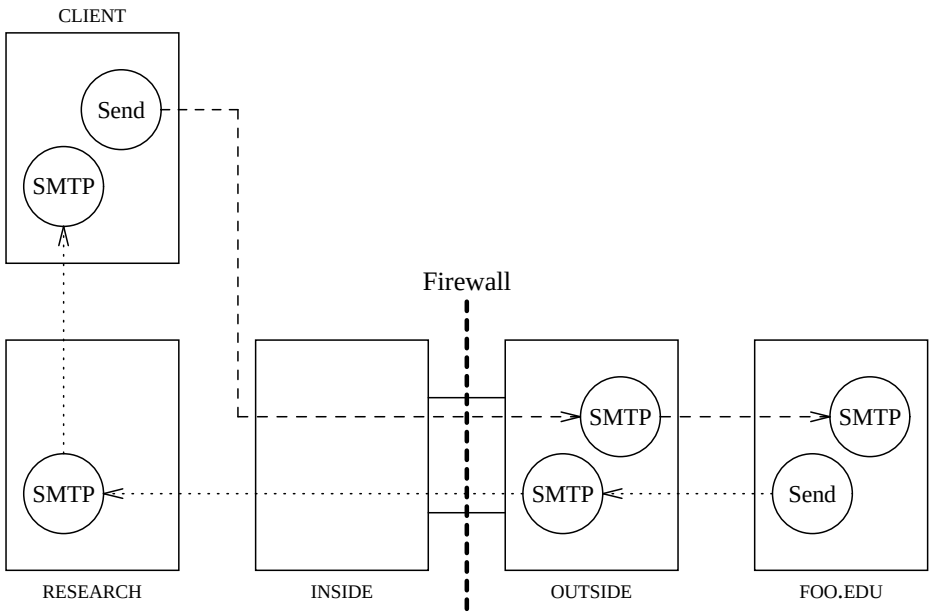

Figure 4.2: Passing mail through the firewall. The dashed lines show outbound mail; the dotted lines show inbound mail. The circles show the sending and receiving SMTP processes.

---

outgoing mail gets stored on OUTSIDE, if only for a few moments. (Gateway transit time is typically under two minutes.) We also have a preexisting, well-known internal mail hub named RESEARCH. This machine is distinct from INSIDE, though it certainly doesn't have to be. Still, we prefer it that way; INSIDE does some sensitive things, and we don't really want to expose it to the vagaries of the *uucp* subsystem that RESEARCH has to run.

We wish to have OUTSIDE deliver incoming mail to the SMTP port on RESEARCH. We also wish to have any internal machine deliver outgoing mail directly to OUTSIDE without going through RESEARCH. We grant both of these wishes using fixed TCP tunnels through INSIDE:

- Configure the mailer on OUTSIDE to deliver incoming mail to INSIDE on port *26*! We've picked this port number at random and will dedicate it to relaying SMTP connections from OUTSIDE through INSIDE to RESEARCH.
- On CHOKE,

action	src	port	dest	port	flags	comment
allow	OUTSIDE	> 1023	INSIDE	26		
allow	INSIDE	26	OUTSIDE	> 1023	ACK	

  which allows the incoming connection.

- On INSIDE, add to `/etc/inetd.conf`

```
port26 stream tcp nowait daemon /v/gate/tcpd /v/gate/relay tcp!research!25
```

  and configure the TCP wrapper to permit the connection only from OUTSIDE. (In this case, it doesn't matter: Who cares if an internal user launders a connection to RESEARCH?) Don't forget to add `port26` to `/etc/services`.

For a direct connection to OUTSIDE from any inside machine, add this entry to INSIDE's `/etc/inetd.conf` file:

```
smtp stream tcp nowait daemon /v/gate/relay relay tcp!outside!25
```

and let it through the router:

action	src	port	dest	port	flags	comment
allow	INSIDE	> 1023	OUTSIDE	25		
allow	OUTSIDE	25	INSIDE	> 1023	ACK	

That is, callers to INSIDE's SMTP port will speak directly to OUTSIDE. Mail is not spooled by INSIDE. Our mail setup is shown in Figure 4.2.

Notice that if our router doesn't have the "established" keyword, or if it doesn't work, then we already allow all TCP connections to and from the nonprivileged ports between INSIDE and OUTSIDE. In other circumstances, this would be a serious hole, but here it is not so bad. The major restriction on OUTSIDE's access to INSIDE's services is enforced by INSIDE, not the router. CHOKE's vital role is to restrict the internal machines OUTSIDE can access, not the services. Routers are very effective at host-based restrictions. Also, since INSIDE will have no extraneous users, there will be no one to exploit the high-numbered ports.

4.5.2 Incoming *Telnet*

Incoming authenticated logins are provided via the *telnet* service. The remote user *telnet*s to OUTSIDE and logs in with username *guard*. This uses *relay* to connect the user to the authenticator program on INSIDE. If the authentication succeeds, the guard program on INSIDE calls *rlogin* to connect him or her to the desired host.

Remember that we are authenticating users with some sort of hand-held device, called a *token*. Passwords are not used here. We want the user's proper response to be different for each login, so an eavesdropper can't learn how to do it.

Unfortunately, *rlogin* connections can result in a password prompt, if the user has not pre-authorized connections from INSIDE. If you wish to guard against this, have the authenticator program attempt an *rsh* first; the *rlogin* should be done if and only if the *rsh* succeeds without asking for a password.

Telnet vs. login

Telnet assumes that the caller should be connected through *login*. This is certainly its standard use. But in our case, *login* is just an extra bureaucratic step. We are doing our own authentication further down the line.

In fact, the use of *login* actually weakens our gateway. If all our maintenance is performed through the console, we don't need *telnet* access to OUTSIDE at all. The presence of *login* gives external folks an unnecessary shot at logging in. There have been problems with *login*, especially on systems where environment variables can be assigned values at login time, and we use the environment to pass information around.

The obvious response is to plug a *relay* call in where the *telnetd* call would go, so the bytes simply fly through to INSIDE. Then we'd need *telnetd* and *login* on INSIDE, resulting in the same problem. Our inclination at this point was to relay directly to the authorization program. But then we had character echoing and other terminal setting problems, because there was not a program to process the *telnetd* protocol.

One needs to have a *telnetd* at the other end of a *telnet* session to execute the server's part of the *telnet* protocol. We ended up adding a parameter to *telnetd* that lets us specify the desired "login" program. Then *telnetd* can perform its terminal setting functions, but we can avoid the unnecessary *login* invocation. We've since found other uses for the modified *telnetd*, such as the gateway services menu described in Section 4.5.4.

Another approach is to replace the *login* command itself. You can make it do what you want, and install strong authentication. But take care if you intend to permit local logins via this mechanism; on some platforms, *login* does some unusual things, such as enforcing license restrictions on how many users are allowed to connect at once.

We install the service:

- If you don't have a modified *telnetd*, install user *guard* in `/etc/passwd` on OUTSIDE:

  ```
  guard::14:14:security guard:/v/gate/usr/guard:/v/gate/guard
  ```

- Turn on the *telnetd* service. Here, it is a good idea to call the TCP wrapper, since its logging can be useful:

  ```
  telnetd stream tcp nowait root /v/gate/tcpd /usr/etc/telnetd
  ```

  If you do have a modified *telnetd*, this line should simply be a call to *relay* (though the TCP wrapper logging is still useful).

- `/v/gate/guard` is a simple shell script:

```
#!/bin/sh
/v/gate/relay tcp!inside!666
```

- Permit connections through the CHOKE:

action	src	port	dest	port	flags	comment
allow	OUTSIDE	> 1023	INSIDE	666		
allow	INSIDE	666	OUTSIDE	> 1023	ACK	

- Add the service on INSIDE:

```
port666 stream tcp nowait root /v/gate/guard guard
```

  Add `port666` to `/etc/services`.

Of course, we haven't mentioned the actual guard program and its database. The TIS toolkit contains a suitable version. The basic program is very simple: we wrote the original in a morning. The hard parts are learning how the hardware token encrypts the challenge, and figuring out how to keep the key database secure. The latter includes expiration of accounts, logging, and provisions to shut down an account when too many unsuccessful attempts occur.

In this arrangement, the database could be kept on INSIDE, which is a secure machine. In our environment we use the Plan 9 [Pike *et al.*, 1990] authentication server.

If an authentication server sounds too complicated, here's a shell script that makes a passable skeleton for an authentication service:

```
#!/bin/sh

read username
challenge=`makechallenge`
echo    "Challenge: " "$challenge"
echo -n " Response: "
read response

if [ verifychallenge "$username" "$challenge" "$response" ]
then
        echo OK
else
        echo NO
fi
```

A program is a better choice.

4.5.3 Proxy Services

Outgoing TCP calls must also pass through the firewall. Our outgoing *proxy* circuit-level service is easy to install:

- On INSIDE, relay connections to OUTSIDE. We use port 402 for *proxy* service. Add port 402 to `/etc/services` and add to `/etc/inetd.conf`:

```
proxy stream tcp nowait daemon /v/gate/relay relay tcp!outside!proxy
```

- Permit the outgoing connections on CHOKE:

action	src	port	dest	port	flags	comment
allow	INSIDE	> 1023	OUTSIDE	402		
allow	OUTSIDE	402	INSIDE	> 1023	ACK	

- Add the service on OUTSIDE:

```
proxy stream tcp nowait daemon /v/gate/tcpd /v/gate/proxy
```

- Add port 402 to OUTSIDE's `/etc/services`.
- Don't forget to limit the connections to this service on OUTSIDE from INSIDE via the TCP wrapper. Otherwise, carefully equipped outsiders could launder connections to other outside machines through your proxy service!

For single-machine application gateways, one could use *socks* to implement the TCP circuit gateway on a single-machine (Plan A) gateway. As of this writing, *socks* won't work if the external Internet name service is not available to internal machines, because it directly replaces socket calls that use IP numbers, not names. It would not be hard to extend it with an `Rgethostbyname` routine that looked up external names on the client host.

It is not difficult to arrange to bring external name lookups to internal machines. A large company with an application gateway usually has its own root name server for internal machines. This can be extended to look up external addresses for internal users (see Section 3.3.4). Many gateway designs adopt this approach and, again, *socks* requires it. But this is harder than it seems. You have to watch for contamination of the internal DNS with addresses of unreachable root servers. We are troubled enough by this problem that we have not implemented it in our gateway, and are reluctant to recommend it to others.

4.5.4 A Gateway Service Menu

We do not install general user accounts on our gateways. We only give these accounts to system administrators with a clear need for them. This lack of direct users is an important security feature. User-controlled security features like `.rhosts` and user-selected passwords tend to be a weak link in a system's security.

```
$ telnet proxy
This is the new inet.  Authorized use only.

stty erase ^H kill ^U intr ^C nl tabs
 9:50pm  up 1 day, 9:30,  1 user,  load average: 0.26, 0.20, 0.24

This is the research gateway services menu.
Watch this space for gateway information.  Please send
complaints and comments to ches@research.att.com.

Enter 'help' for command list.

research gateway> help

        ping <destination>
        traceroute <destination>
        dig <parameters>
        telnet <destination> [port]
        quit

research gateway> quit
```

Figure 4.3: Our gateway services menu.

But the average user, or internal system administrator, does have a legitimate need to access some external network functions. While we could write versions of these that used the proxy library, we have found that nearly all these needs can be met with a simple captive shell script on the gateway machine, accessible from the inside.

Figure 4.3 shows the menu our users see. The services include *ping* and *traceroute* to probe for network problems, *dig* for name service lookup and debugging, and *telnet*. This last is a handy service for internal users who don't have a converted version of *telnet*. In fact, if outgoing *telnet* service is all that is desired, the installer can skip the *proxy* gateway altogether and just use this.

On single-host gateways, one could add an FTP service to the menu using an FTP gateway like Digital's or the *Toolkit* from the Trusted Information Systems Toolkit. It is harder, though not impossible, to make an FTP service like this through a double-machine gateway.

The circuit connections for our service menu are easy:

- On INSIDE, relay *telnet* connections to OUTSIDE. We need to invent a private TCP port for this link, since the *telnet* service on OUTSIDE is already serving the outside world with guard service:

```
telnet stream tcp nowait daemon /v/gate/tcpd \
        /v/gate/relay tcp!outside!667
```

- Permit the outgoing connections on CHOKE:

action	src	port	dest	port	flags	comment
allow	INSIDE	> 1023	OUTSIDE	667		

- Add the service on OUTSIDE:

```
port667 stream tcp nowait root /v/gate/tcpd \
        /v/gate/serviced -l /v/gate/serviced.sh
```

  In this case, `/v/gate/serviced` is a link to *telnetd*. We had to do this because of how the TCP wrapper works: It uses the real daemon name as the search key for access control. By using an alias, we can configure the TCP wrapper to limit this service to originate from INSIDE. Notice also that we used our modified *telnetd* daemon (described earlier) to avoid involving *login* in this operation. We have to run *telnetd* as *root*, so that it can allocate a pseudo-tty.

As usual, careless shell programming, or those wretched shell escapes (those internal commands that some programs have that give the user a shell prompt) from assorted otherwise-innocuous programs, offer the largest security threats to this service. It might be a good idea to implement this service in a `chroot` environment on OUTSIDE in case someone escapes from a menu program. None of these services needs to run as *root*.

Our *serviced.sh* script is shown in figure 4.4.

4.5.5 Anonymous FTP

It is pretty easy to set up an anonymous FTP service. For the concerned gatekeeper, the problem lies in selecting the version of *ftpd* to install. In general, the default *ftpd* that comes with a system lacks sufficient logging for the paranoid gatekeeper. The gatekeeper wants the sessions logged. The file providers want the file accesses logged to keep tabs on the public interest in the files they publish with FTP. Versions of *ftpd* range from inadequate to dangerously baroque.

We made our first *ftpd* by modifying a basic BSD version available on the network. We added a few *syslog* calls, and ripped out most of the login features, making *anonymous* (a.k.a. *ftp*) the only permitted login. (Historically, there have been bugs in the FTP login code. Why do people try to replicate this vital function in different places, rather than performing it once in common code?)

Our latest *ftpd* starts from similar roots. This time, though, we ripped out all of the login code and all of the code that required any *root* privileges to execute. The body of the server does nothing but execute the basic protocol. We also deleted code of dubious merit, such as commands to let the user change file permissions or the `umask` setting. (Those changes were independent of the use we make of the permission scheme, as described later. Rather, they derive from elements of our basic philosophy: The system managers should set the security policy, and hence the access rights, not anonymous users on the Internet.)

```
#!/bin/sh

stty erase "^H" kill "^U" intr "^C" tabs
echo stty erase "^H" kill "^U" intr "^C" tabs
/usr/ucb/uptime
echo
trap "" 2

cat /v/adm/motd
echo "Enter 'help' for command list."
while true
do
        trap 'continue' 2
        echo "research gateway> \c"
        if read line
        then
                case "$line" in
                "")     continue;;
                esac

                set -- $line
                case "$1" in
                help) echo "  ping <destination>"
                        echo "  traceroute <destination>"
                        echo "  dig <parameters>"
                        echo "  telnet <destination> [port]"
                        echo "  finger <destination>"
                        echo;;
                ping) shift
                        /usr/etc/ping $*;;
                traceroute)
                        shift
                        /v/bin/traceroute $*;;
                dig) shift
                        /v/bin/dig $*;;
                telnet) shift
                        /usr/ucb/telnet $*;;
                finger) shift
                        /usr/ucb/finger $*;;
                exit) break;;
                quit) break;;
                *) echo "Unknown command - $line."
                        echo "Enter 'help' for command list";;
                esac
        else
                break
        fi
done
exit
```

Figure 4.4: Our gateway services menu program.

Login is handled by a series of separate programs. Access to the individual programs is regulated by the TCP wrapper program. Since only anonymous FTP is offered to the outside world, the FTP login program that is executed implements only anonymous logins, and does nothing else. It immediately `chroot`'s to `/usr/ftp`, and then executes the real daemon. If someone does escape from *ftpd* somehow, they are not in a very interesting place, and the login program itself is small enough that we are confident it works. (We have contemplated dispensing with the anonymous login program entirely. Why go through the ritual of asking for information that you don't need if the client is honest, and that you can't verify if the client is dishonest? Of course, some people use the voluntarily supplied electronic mail addresses to notify people of software updates, etc.)

We use the ordinary UNIX system permission mechanisms to defend against our system being used as a transfer point. The `umask` for outside users is set to 422_8. That is, files created by anonymous users are created without read permission by the owner: *ftp*, or write permission by *group* or *other*. In other words, the daemon will create files that it does not have permission to read. Execute and write permission is permitted, so that usable subdirectories can be created. Internal users access FTP via one of the firewall gateways; this makes it easy to configure the TCP wrapper to let AT&T users run under a different anonymous account, with a more conventional `umask`. Thus, they can read the files via the "group" or "other" permission bit. Similarly, files they create are world-readable, and thus available to outsiders. No complicated mode-checking need be done by our server.

We do need to be careful about installing anything else on OUTSIDE, such as *gopherd* or an FTP-by-email server, that might itself initiate an FTP session. Such a session would appear to be from the same machine as the proxy relay, and hence would appear to come from inside. A number of solutions suggest themselves. We could have such servers use a special range of port numbers when initiating FTP connections. Our proxy gateway would use a disjoint range, thus permitting filtering to take place. Or we could use a special password for anonymous FTP logins, requesting *less* privilege. Best of all, we could run these servers (including FTP, if desired) on a separate machine from the proxy gateway, which would totally eliminate the problem.

Permanent files are deposited by insiders who connect to port 221 (a more-or-less random number) on the gateway. This port number uses a different login program, one that does require a user name and password. It still does a `chroot`. We could install individual maintenance accounts, but have never felt the need to do so in our comparatively small community of users who publish via FTP.

Most implementations of *ftpd* need to retain *root* privileges so that they can bind to port 20 for the data channel, as is required by the protocol. Ours invokes a small, stupid program named *port20*. It checks to see if file descriptor 0 is a TCP stream bound to port 21 or port 221. If it is, the program binds file descriptor 1 to port 20 or port 220. In other words, if the program is the FTP server, it is entitled to use the server's data channel. Other necessary rituals, such as issuing the `SO_REUSEADDR` call, are done by the caller; it, being unprivileged, has less at stake, and the privileged program should be as small as possible.

The actual setup of an anonymous FTP service is described well in the vendor manual page, though some vendors are rather careless about security considerations. Three caveats are worth repeating, though: Be absolutely certain that the root of the FTP area is not writable by anonymous

users, be sure that such users cannot change the access permissions, and do *not* put a copy of the real `/etc/passwd` file into the FTP area (even if the manual tells you to). If you get the first two wrong, an intruder can deposit a `.rhosts` file there, and use it to *rlogin* as *ftp*, and the problems caused by the third error should be obvious by now.

4.5.6 The MBone

The MBone provides multicast audio on the Internet. It's used for IETF meetings, and for the *Internet Talk Radio*. Lots of folks inside our firewall want to participate. However, MBone uses UDP, so our normal proxy mechanisms do not work. Instead, we use a special program *udprelay* to copy external MBone packets to selected destination hosts and ports on the inside. An internal redistribution tree is used to limit bandwidth consumption.

A more complete solution involves extending the MBone to inside networks, including use of IP encapsulation and DVMRP. The danger comes from abuse of these mechanisms—which after all, constitute a fairly general packet routing and relay device—to deliver other sorts of packets. Although our solution is not yet final, we intend to filter and modify incoming MBone packets. First, only multicast addresses and the proper protocols will be passed. Second, the UDP port number will be changed to one residing in a range that we believe is safe. Third, the session directory packets will be modified to contain the new port numbers.

There are hooks in the session directory program that can be used to inform the gateway when multicast reception has been requested for a given port. Although in principle this a good idea, it does raise the troubling notion that a user could abuse the mechanism to open the firewall for any UDP ports.

Outbound traffic raises its own set of concerns, including the possibility of inadvertent "bugging" of our offices via a microphone installed on a workstation. Anything said near that workstation will be relayed via the MBone, to anyone who is listening.[1] Similar concerns would apply to any outbound video feed, electronic conferencing system, etc. Perhaps the gateway should block outgoing traffic, but provide a means for temporarily enabling transmission by the user on a per-station basis.

4.5.7 X11

X11 poses problems. From a security standpoint, the X11 user simply gives away control of his or her screen, mouse, and keyboard. From a user's point of view, X11 is the basic, essential software platform. It often makes good business sense to use X11, even if there is no secure basis to trust it.

So if an X11 terminal is on the Inside, how can the user give away a little access without giving away the store? We developed a pair of programs called *xp11* and *xgate* (Figure 4.5). The former is invoked by user processes; the latter runs on a well-known port on the gateway machine. *Xgate* listens to a port numbered 6000 plus a random integer, and returns that port to *xp11*. *Xp11* generates and prints an X11 `$DISPLAY` setting suitable for connecting to that port. The user

[1] See CERT Advisory CA-93:15, October 21, 1993.

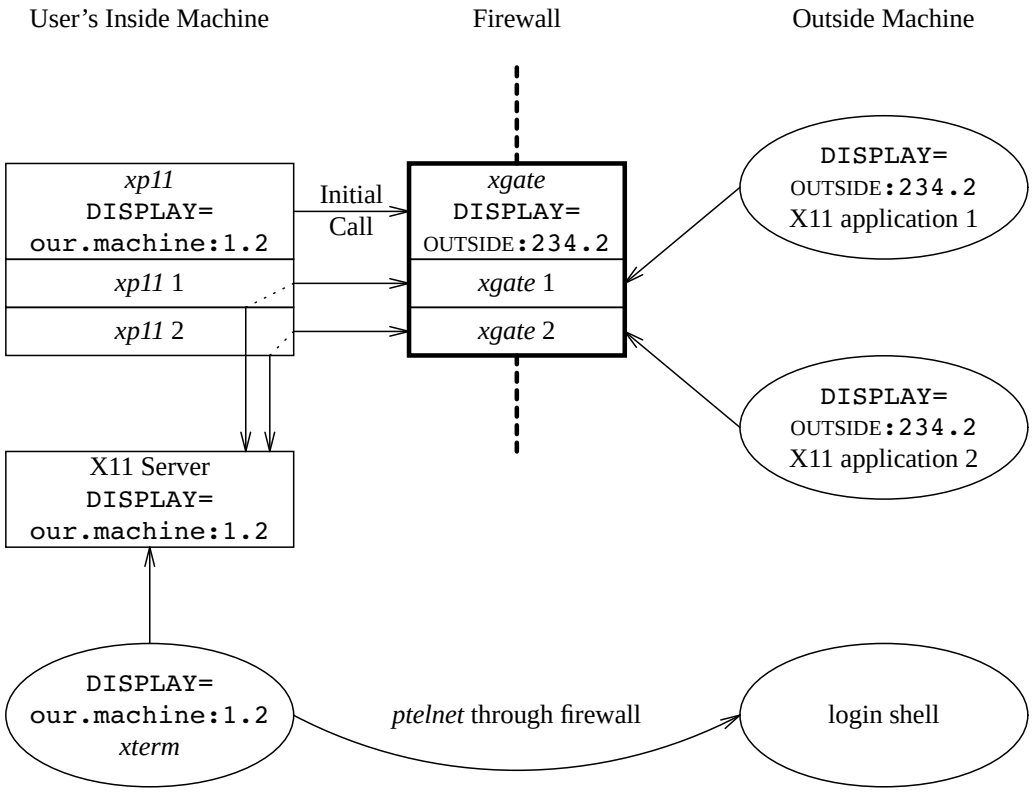

Figure 4.5: A call diagram for *xp11*. Arrows show the direction of each call.

must send this display value to the external X11 application, which then has an external port to which it can connect. When a connection request comes in, *xgate* creates a new random socket, and tells *xp11* about it. A small window pops up (we use *xmessage* to do this; why reinvent yet another wheel?); if the user gives permission, *xp11* calls both the new *xgate* socket and the actual X11 server, and relays the bytes back and forth. This design avoids complicated code to multiplex many different circuits and also avoids having to handle incoming calls from OUTSIDE.

We use a similar program, *px11*, for situations where the X11 server is on the outside and the application is on the inside. It is invoked as

```
px11 real-$DISPLAY [ application [args...]]
```

where "*real-$DISPLAY*" is the name or IP address of the actual server, and "*application*" is the program you want to run, by default a shell. It operates by invoking the application in an

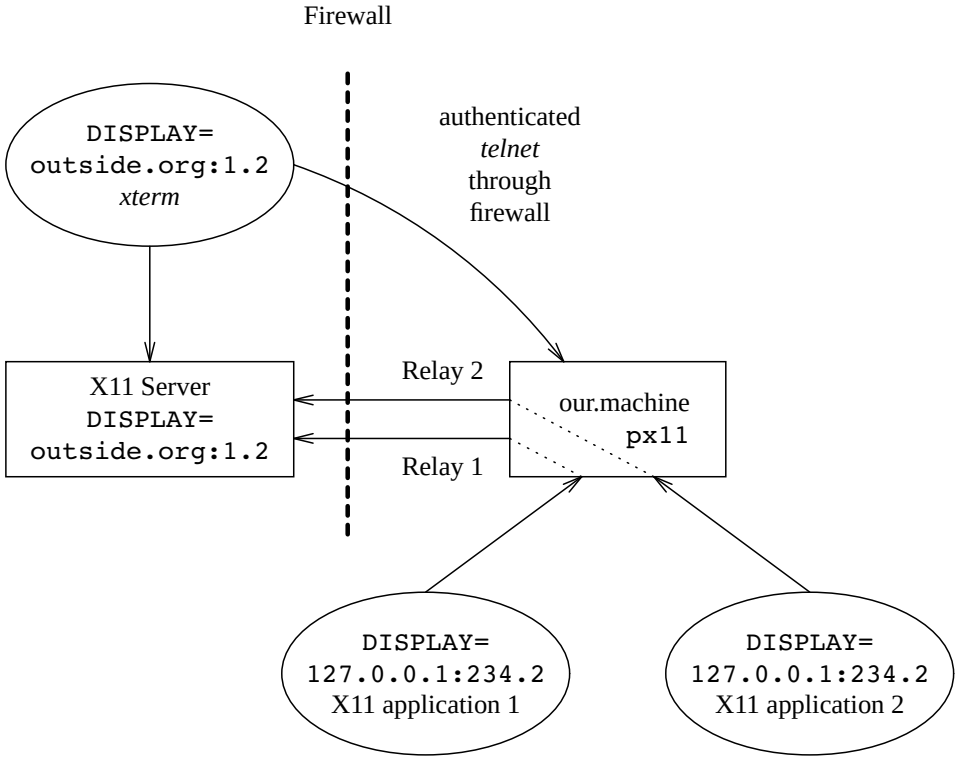

Figure 4.6: A call diagram for *px11*. Arrows show the direction of each call.

environment where `$DISPLAY` points to a socket opened by *px11*, and then by opening a proxy TCP call to the outside server for each incoming connection (Figure 4.6). In effect, it and *xp11* operate as specialized versions of *relay*.

Xforward, a program similar to *xp11*, has been implemented at Digital Equipment Corporation [Treese and Wolman, 1993]. Its behavior is similar to our program's: You provide it with a list of authorized client machines, and it pops up a confirmation window for each request.

Even with these solutions, X11 is still dangerous. The "security" only works down to the host level, which means that another user on the external server machine may be able to step in first and do harm. Also, if an X11 session is hijacked, more extensive control and bugging of the terminal is possible. We have contemplated an active X11 filter on the gateway that understands the zillions of X11 primitives and only permits the safe and authorized ones through. Unfortunately, by its nature such a program would be quite complex, so we would have no reason to trust it.

4.5.8 WAIS, WWW, and Friends

These services are relatively new and obviously quite attractive to users. They represent a significant step toward the construction of an automated Internet-based "cyberspace." The early servers have scared us—we are easily scared by big new programs. Our first response to supplying the world with our own service has been to install them on a dedicated sacrificial machine outside our gateway.

The problem is that the servers are too powerful. With *httpd*, a common WWW server, the really interesting answers are typically generated by shell or *perl* scripts. Not only do we not trust the scripts, we do not want such powerful interpreters lying around.

Similar concerns apply to *gopherd*, especially if the server shares a file system subtree with FTP. In that case, an attacker can deliver harmful shell scripts via FTP, and then try to persuade *gopherd* to execute them. Our impression is that not much persuasion is needed.

More recently, our Z39.50 [ANSI, 1988] developer (Z39.50, the ancestor of WAIS, is a library information protocol) has convinced us that he can supply his service without giving away the store. We plan to tunnel the service to his internal machine in the same way as the printer example discussed in Section 4.4.2. Even so, his server will likely run in a `chroot`'ed environment.

Giving our internal users access to these Outside services has involved converting the *xmosaic* program to use the proxy library. This has worked fairly well, but the maintainers complain that the modified program is (quite naturally) not supported by the original developers. Modifications for *socks* are widely available, but we are unable to provide *socks* support (see Section 4.5.3). The newly announced *xmosaic* gateway function—in effect, an application-level gateway for it—may prove to be the ultimate solution.

In any case, the modified *xmosaic* is potentially dangerous. Some of these new database services are, in effect, MIME, and have the same set of concerns. The developers are grappling with these issues.

4.5.9 Proxy NFS

As mentioned earlier, we have developed a proxy version of NFS that is not only safer, but also lets us use NFS through the firewall. We use it for auditing our disks and we may use it to allow insiders to make files available for anonymous FTP. To understand how our proxy version is implemented, it is first necessary to know a bit about how NFS works.

When a system wishes to access a remote file system, the local *mount* command contacts the remote mount daemon, *rpc.mountd*. After doing the usual (inadequate) access control checks, it passes back the proper IP address, UDP port number, and file handle for the root node of the requested file system. The *mount* command then passes these items to the kernel.

What is important to realize here is that the kernel neither knows nor cares exactly what is at the other end of that UDP port. All that matters is that it speak the NFS protocol; it need not be a real disk on another machine. Examples include the automount daemons [Callaghan and Lyon, 1989; Pendry, 1989] and an encrypted file system [Blaze, 1993].

We use the same trick. On the NFS client—the machine that actually issues the `mount` system call—we run a daemon on the loopback interface (127.0.0.1). It receives the NFS requests,

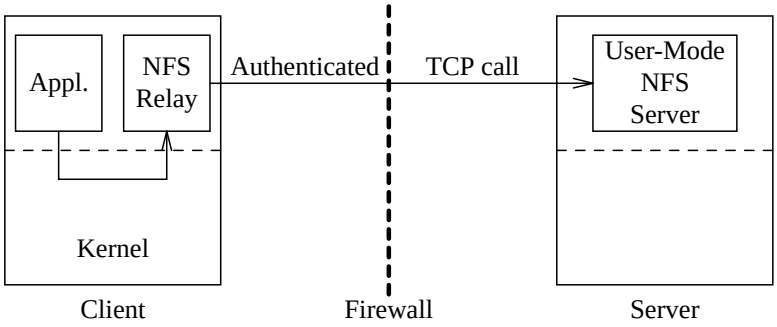

Figure 4.7: A proxy NFS connection.

packages them, and ships them out via TCP to the actual server.

The server could perform the other half of the process. That is, it could take the request, and use its own loopback interface to forward the message directly to its local NFS server port. But that would require that the machine offer NFS services, which we are loathe to do. Instead, we pass it to a *user-level* process that emulates NFS via ordinary file system calls: `open`, `close`, `read`, `write`, etc.[2]

Going through the file system has many advantages. One is already obvious: We do not need to run NFS on the gateway. Another advantage is more subtle: we can take advantage of the protection mechanisms that the UNIX system has to offer, such as `chroot`, user and group names, and file modes. These traditional kernel-level protections are well-known and probably more trustworthy. If our user-level server is compromised, we lose control of the data it offers, but when it is run in a `chroot`'ed subtree, nothing else is exposed. This gives us a much finer granularity of protection than does NFS's entire-file-system approach. Similarly, the server need not even run as *root*, if a single permission level would suffice. This lets us protect files that live in an exported subtree.

For protection, we can encrypt the TCP circuit if we wish. The use of user-level programs at both ends makes that simple. We do use cryptographic techniques to generate the file handles, protecting us against guesses. (We are guarding here against attackers who send bogus NFS requests to our server process.) The file system emulator incorporates a user identifier map, which facilitates interorganizational use: There is no need for both ends to use the same numeric user IDs.

Note that we have said nothing about which end initiates the TCP call. Generally, it will be the side doing the `mount` call, but it need not be. If it is easier to pass through a firewall in the other direction, the server can call the client. (But we do incorporate challenge/response authentication into our calling module.)

[2]The user-level NFS code was developed by Norman Wilson, and was inspired by an earlier user-mode file system written by Peter Weinberger [Rago, 1990].

4.5.10 Installing NTP

Both INSIDE and OUTSIDE need to know the correct time. On OUTSIDE, we installed NTP, and had it talk to a carefully chosen set of servers. Using well-known time sources is a good idea; it's harder to tamper with a machine that lots of people rely on, since the attack would be more likely to be noticed. We also configured our NTP daemon to reject requests from other sites.

We run NTP on INSIDE, too. It speaks NTP to OUTSIDE and to our internal NTP servers. The time derived from the inside machines is exported back to OUTSIDE; this provides an additional measure of protection against time-spoofing.

The following rules have to be added to CHOKE:

action	src	port	dest	port	flags	comment
allow	INSIDE	123	OUTSIDE	123	UDP	
allow	OUTSIDE	123	INSIDE	123	UDP	

Note that we only permit the NTP daemons themselves to speak; NTP queries and control messages from random ports are blocked.

4.6 Protecting the Protectors

There are a few things our gateway can't do for itself; it needs a bit of extra protection. To accomplish this, we asked our regional network provider to install a short, simple filter list on our router port. The threats blocked are, for the most part, low grade, but they're still worth avoiding.

The biggest problem is that much of our proxy functionality is protected by the TCP wrapper program, and it, in turn, relies on the source IP address. We thus have the router filter out packets that claim to be from the Inside net, since such a packet can't possibly be legal. Similarly, we delete packets purporting to be from 127.0.0.1, the loopback address.

Next, we block anything heading toward our *syslog* port, mostly to avoid denial-of-service attacks on it. We could add that check to the daemon itself, especially with the router blocking forged source addresses.

Finally, as noted earlier, we block anything from the outside heading toward the CHOKE router.

This simple ruleset is probably reliable. Filtering whole networks or hosts is more reliable than trying to filter on the service level.

action	external	port	local	port	comment
block	127.0.0.1	*	*	*	no loopback pretenders
block	*	*	*	514	no connects to syslog
block	*	*	CHOKE	*	nothing goes to choke
block	LOCAL-NET	*	*	*	no local net packets from outside
allow	*	*	*	*	all else is ok

4.7 Gateway Administration

> The fool says "Don't put all of your eggs in one basket," but the wise man says "put all your eggs in one basket *and watch that basket!*"
>
> —PUDDIN'HEAD WILSON'S CALENDAR

The external host OUTSIDE is fully exposed to the perils of the Internet. One can take a number of steps to keep an eye on that machine and watch for trouble.

4.7.1 Logs

We've recommended that the external machine keep extensive logs, but these logs can pose a problem for the system administrator.

They can take *lots* of space. Our research gateway typically generates 20 to 30 MB of logs daily. The logs should reside in their own disk partition, with plenty of space for unusual activity. There is no excuse to stint on disk space: Give it gigabytes if necessary. We now get disk space at less than $400 per gigabyte.

Keep your logs for at least a week. Most mail problems are reported within that time. We have found it useful to extract key information and save that indefinitely. This provides interesting statistical information, and can help you detect long-term slow attacks.

Use daily scripts to extract essential information. Our definition of "essential" has become much more restrictive over time as the weight of probes has buried us. An essential event might be an attempt to *rlogin* as *root* or to *tftp* the `/etc/passwd` file. A daily mail message with a single line for each key incident on the previous day isn't too bad. In most cases you can probably detect interesting or dangerous attacks better than the AI-based approach some people are espousing.

Log files have a lot of redundancy and are easily compressible. If you're short on space, try using a compression command on older log files. The trade-off, of course, is that it is harder to scan a compressed file.

The logs don't have to be stored on the external machine and it is probably better not to. Consider implementing a *drop-safe* for logs: a write-only port on the inside backed by a big disk. If this port is used to store the logs in real time, a successful hacker will be unable to delete your logs and cover his tracks. The syslog facility is quite useful for logging in general, and may be used for sending the logs to the drop-safe.

4.7.2 File Integrity

As usual, it is a good idea to back up the files on your gateway machine. Since you presumably do not have a significant user population on that machine, most files won't change very often. We like to copy whole file systems to free partitions on hot spare disks. These extra disks help us get the vital gateway machine back on line quickly when one of our disks dies, which is the chief failure mode on our gateways.

Backups should also include some off-machine backup, either to another less-accessible computer over the network or onto a tape of some sort. Though these backups are infrequent, they should be kept for a long time. If you suddenly discover that your computer was compromised nine months ago, do you have an older backup to fall back on? (Remember the day 0 backup we suggested earlier?)

It is very useful to monitor changes in the contents of important directories and files. In a quiet moment late each night, create a list of all the set-user-id files owned by *root*, *bin*, and *uucp*, and detect and report changes to the list. Report changes to any important directory, especially those that contain programs that *root* may use at any time, including system startup and *cron* jobs. Figure 4.8 shows how we generate our checklist and check it.

If you can afford to run your checks more often, do so; you may catch a hacker after he or she has broken in, but before everything has been reset to an innocuous state.

We use our proxy NFS to run the checks from the inside. One can run these checks on the gateway computer itself, although a successful, skilled hacker could corrupt these checks or modify the database of known-valid checksums. (An even more subtle person could corrupt our network file system server and achieve the same result, or point the mount point at an unmodified copy of the file system. This is much more work than most hackers are willing to try, and such meddling increases the risk of detection. If the hackers start doing such things, you could install a dual-ported disk and read it directly.)

These file checks are well worth doing. Most of the file changes are made by friendly fingers or glitches in the machine. In any case, one can hunt down and understand the source of these changes and correct them before the Bad Guys notice and take advantage of them.

The *tripwire* program [Kim and Spafford, 1993, 1994a, 1994b] takes a good shot at solving these problems and more. You should consider using it.

4.7.3 Other Items

A number of other things should be considered:

- How do administrators get access to INSIDE and (especially) OUTSIDE? Apart from the console access, we run a vanilla *telnetd* on a variant port, suitably protected by the TCP wrapper program.

- How are new binaries installed on OUTSIDE from the inside? Does one use FTP to transfer them? Our proxy NFS arrangement would be suitable.

- How do internal users update files in the FTP distribution directory? Our latest solutions involve the use of FTP itself or perhaps proxy NFS.

- How can the internal users be notified of gateway-related news? It's hard in our case, because we have tens of thousands of potential users. We try to keep a mailing list of the major contacts, and our relay and proxy logs contain user/machine addresses, obtained from the proxy protocol.

```
cat     <<! >checklist
/unix
/bin/*
/usr/bin/*
/usr/net/*
/usr/ucb/*
/usr/etc/*
/usr/lbin/*
/usr/new/*
/bsd43/bin/*
/etc/gated.conf
/etc/syslog.conf
/etc/inetd.conf
/etc/passwd
/etc/group
/etc/named.d/*
/usr/spool/cron/atjobs
/usr/spool/cron/crontabs/*
/usr/adm/rotatelogs
/usr/adm/periodic/driver
/usr/lib/sa/sa1
/usr/lib/sa/sa2
!

eval ls -d `cat checklist` >filelist.new
echo
echo "*** changes to the list of files checked:"
diff filelist filelist.new

echo
echo "*** changes in files:"
>>sum.new
for i in `cat filelist`
do
echo "$i `hash2.0 4 256 <$i`" >>sum.new
done

diff sum sum.new
```

Figure 4.8: The *hash2.0* uses the 4-pass, 256-bit output version of Merkle's *snefru* algorithm [Merkle, 1990a] to compute the checksum. Don't use *sum* instead. Many hackers have tools that let them tailor the output of the *sum* command. Other checksums would be more secure; see Section 13.1.7. One might be tempted to just use a simple secret checksum, like a variation on the standard *sum* command. This is a bad idea because the secret may get out.

- Logs should be rotated. We keep them on the machine for a week, and copy some of them off to Plan 9's WORM drives forever. The idea is to make these gateways as maintenance-free as possible, and growing logs can creep up on a system administrator.
- What kind of hacking reports are generated for the system administrator? Does CERT get a copy, too?

4.8 Safety Analysis—Why Our Setup Is Secure and Fail-Safe

As we have stated repeatedly, it is not enough for a gateway to *be* secure. You must also *know* that it is secure and that you have configured it properly. You should also see what will happen if your analysis is wrong—i.e., if your gateway is not secure. Can the Bad Guys get in? How far can they get?

We have some ground rules for performing our analysis. We trust simple things like files, programs, or very short router filter rules. We don't trust complicated things. We're happier when we have all the source code, because we can then assess the complexity of any privileged sections of code.

Here are the services we run and our analysis of them:

FTP: Protected by early `chroot`; most of it does not run privileged.

SMTP: Uses the *upas* mailer, which we believe to be more secure than *sendmail*; talks only to *upas* on RESEARCH.

telnet: Connects directly to INSIDE, but only to a small program that does challenge/response authentication.

printer: Protected by the TCP wrapper; connects only to a trusted destination.

MBone: Simple program; just relays UDP packets to known destinations. Future expansion worries us, as does the outgoing channel. Multicast has important security implications.

proxynfs: Protected by the wrapper; access only from INSIDE. Runs in a `chroot` partition.

proxy: Protected by the TCP wrapper; access only from INSIDE. Wrapper failures only permit connection laundering.

proxyd: Test version of *proxy*; used for debugging changes and enhancements. Normally it isn't activated.

insidetelnet: Protected by the TCP wrapper; access only from INSIDE. Wrapper failure just gives our harmless proxy services menu.

secrettelnet: Protected by the TCP wrapper; access only from INSIDE. Gives you a real login prompt, but there are no real user logins on the machine.

As you can see, very few services—just FTP, SMTP, *proxynfs*, and *secrettelnet*—give any real access whatsoever to OUTSIDE itself, and two of them are walled off by `chroot`. One of those is protected by the TCP wrapper as well, and *secrettelnet* presents an intruder with a *login* prompt for a system that has exactly *one* real entry in `/etc/passwd`.

Three services are relayed directly to INSIDE. The MBone relay is harmless, as is the print service. The most dangerous one—*telnet*—is protected by a simple but strong authenticator. (Not that that is perfect; we have had problems with that service, though only *after* authentication. We were thus exposed to risks from insiders only.) Our X11 relay is always initiated from Inside, thus obviating the need to expose that service to Outsiders.

The biggest risk here is with SMTP. If the mailer is buggy on OUTSIDE, it is likely to be buggy on RESEARCH as well. A common-mode failure here could indeed endanger us. For that reason, we have contemplated using a different protocol to move the mail from OUTSIDE to RESEARCH, one that presumably would not share the same hypothetical flaws. Failing that, there are few compelling reasons even to spool the mail on OUTSIDE; we could just use *relay* to pass it directly to the inside mail relay, though there is some performance benefit to doing mailing list expansion on the outside.

If, despite these precautions, OUTSIDE were to be compromised, the attacker would still have a daunting task. The CHOKE router provides access to very few inside services, and there is no way to connect to it from any external machine. Nor can packets be routed through it; we do not advertise internal routes to the outside, nor do we permit source-routed packets.

Suppose the worst happens. We have a few other special-purpose machines on our outside net; certainly, one of them could be compromised, and an intruder could use it to monitor our gateway network, perhaps picking up the *root* password to OUTSIDE. (This would only occur if we used the *secrettelnet* service to login. The console login doesn't travel over the external network.)

It wouldn't do the hacker much good: *There's no way to use it from outside.* The administrative login mechanism uses the TCP wrapper, blocking any access from outside. Even if that mechanism were to be defeated, the ability to log in to OUTSIDE, even as *root*, gives you just that one machine. Further entry to Inside is restricted by the challenge/response authentication mechanism on INSIDE. At most, the hackers could try to subvert the relay program to try to capture a legitimate session. But that's a risk we're willing to take for any connections from the outside; the subverted machine could as easily be a university machine as OUTSIDE. *Our exposed firewall has no special privileges except for the right to forward mail and login requests to* INSIDE.

It is not enough to worry about the exterior of a firewall; gateways can be attacked from the inside as well, if the intruder has another path in [Hafner and Markoff, 1991]. Here, too, we believe we are safe: INSIDE is configured almost the same way as OUTSIDE. Again, there are no ordinary user accounts, nor any gratuitous services. In fact, it's more secure; it only relays mail and runs the authenticator. The login service is relayed to the gateway service menu on OUTSIDE and there is no FTP.

Similarly, direct access to the outside via CHOKE must be restricted as well. Most or all of the precautions that are taken on its outside net should be taken on its inside net as well. That is, it should not advertise or listen to any routes, it should have strict access control lists specifying only INSIDE, and the other routers and computers on the inside net should be blocked from speaking to it, preferably at the link level.

4.9 Performance

Performance is not our primary concern. Safety is. But a gateway's performance can dictate its longevity. If it is too slow, internal users are likely to press for other, less secure solutions. So it is important for us to see how fast we can push bytes through.

First we used *ttcp* to measure the throughput from a MIPS Magnum to a MIPS M/120 on our outside net. It ran at about 420 KB/sec. Next, we used a version of *ttcp* that was converted to use our proxy protocol to talk to the same target through the proxy gateway. This version ran at about 210 KB/sec. Obviously, there is a significant loss. But 210 KB/sec is about 110% of the throughput of a DS1 (i.e. T1) line, so we are not particularly displeased.

A more realistic test is how fast we can transfer data to a machine elsewhere on the Internet. A *ttcp* run from OUTSIDE ran at 50 KB/sec—25% of the line speed—while the proxy version to the same destination ran at about 48 KB/sec. Obviously, there is some loss of performance, but it is—to us—well within the acceptable range, and quite likely within experimental error.

Finally, we've had very few complaints of network throughput from our users that could be tied to our arrangement. There have been some complaints about call setup time: the proxy arrangement does involve some overhead. It shows up in FTP's `MGET` command, and in certain *xmosaic* applications. We may be able to improve this with a modified *relay* program that pre-establishes connections from INSIDE to OUTSIDE before a call is received, and uses the `select` system call to avoid the context switching time of our forked processes.

4.10 The TIS Firewall Toolkit

The most complete freely available firewall package is the *TIS Firewall Toolkit*, from Trusted Information Systems (TIS) [Avolio and Ranum, 1994]. It's worth taking a look at how it works and what it provides.

First of all, the Toolkit is just that: a kit. You can deploy the features you need, delete others, and implement new ones. The tools supplied include a program similar in functionality to Venema's TCP wrapper and a fairly general user authentication server, including an administrative interface.

Philosophically, the Toolkit is an application gateway. It can be configured to work in any of the single-machine configurations we have discussed; however, there is no circuit-level proxy protocol for a dual-machine solution such as ours. Authentication of outgoing calls is supported for *rlogin*, *telnet* and ftp; for the latter, the administrator can control the direction of permissible transfers. To connect to outside machines, users connect instead to the corresponding port on the gateway machine; they then specify the actual target desired. If authentication is used on that machine, the user requests a destination of *actual-user-name*`@`ACTUAL.DESTINATION. The intent is to permit connections from unmodified client programs, albeit at the expense of a change in user behavior.

Incoming SMTP traffic is handled by a small front end to *sendmail*; this denies attackers the opportunity to speak directly with a possibly buggy daemon. The existence of a front end also provides a hook for any necessary prefiltering of letter bombs.

Incoming *telnet* calls are directed to an authentication server. Various mechanisms are supported, including several types of one-time password schemes.

Direct TCP tunnels are implemented by a program very similar to our *relay* command.

Anonymous FTP is protected by a `chroot` executed before *ftpd* is run. The subtree made available could contain the `/etc/passwd` file that would be used by the regular *ftpd* login mechanism; this file is distinct from the real `/etc/passwd` file, and controls access to the FTP service only. It is thus possible to implement varying permissions for FTP users, mediated by passwords; anonymous FTP, apart from having its own user ID, could do another `chroot` to a smaller subtree. In practice, the developers use their standard network authentication mechanism, rather than `/etc/passwd`, but the basic approach is applicable to any *ftpd*.

The implementers of the Toolkit have a philosophy similar to ours. Programs are small and simple and, to the extent possible, all components rely on structural security mechanisms such as `chroot`. The documentation includes a safety analysis similar in spirit to ours.

4.11 Evaluating Firewalls

Whether you buy a commercial firewall product or build your own, you must evaluate how well it meets your needs. The same is true for components that you might use in building a firewall, such as a packet-filtering router. Here is a list of questions you may want to ask. Two questions, though, are paramount: Does the solution meet your security needs, and does it meet your connectivity needs?

4.11.1 Packet Filters

- Where is filtering done? Input? Output? Both?
- What attributes can be checked? Protocol? Source port? Destination port? Both?
- Can you filter on established connections?
- How are protocols other than TCP and UDP handled?
- Can you filter on arbitrary bit fields?
- Can you filter routing updates? On input, output, or both?
- Can source-routed packets be rejected?
- How comprehensible is the filter language? Can you control the order of application of the rules?
- How easy is it to configure the rules? What is the user interface like? Menu systems are superficially attractive and are easier to use the first time or two, but they're much less convenient when generating sequences of similar rules than, say, a flat file generated by *awk*.

- What happens to rejected packets? Are they logged? How?

4.11.2 Application Gateways

- What applications are supported?
- Are specialized client programs needed?
- How are the difficult services, such as FTP and X11, handled?
- How open is the platform? Can you add your own application relays? Are the logging, access control, and filtering routines adequately documented?
- How usable is the administrative interface? Is it more difficult to administer your own additions?
- What sorts of logs are generated? Are there data reduction packages provided?
- What sorts of authentication mechanisms are provided for incoming users and outgoing calls? Can you add your own?
- Are any traps or lures provided?
- Can you add your own?

4.11.3 Circuit Gateways

- How portable is the application library? How easy is it to convert your applications?
- What applications have they converted for you?
- Is authentication possible for outgoing calls? Incoming? What types, and can you add your own?
- Is there sufficient logging of outgoing calls?
- How easy is it to build single applications that can talk both internally and externally?

4.12 Living Without a Firewall

What if you can't run a firewall? What should you do then to protect yourself?

First, of course, one should practice all the standard host security measures: good passwords, up-to-date network software, etc. One should do that in any event, of course, but the lack of a firewall means that more care must be taken, for more machines. To avoid any password use, try to install a program like S/Key.

The second part of the answer, though, comes from applying our basic principles. Not all services need be offered on all machines, nor to all comers. Careful analysis may show you a lot of services that can either be turned off or restricted.

Consider TFTP, for example. You may need to run it to permit diskless workstations to boot. But that isn't a facility the rest of the world needs to use. You should restrict it via either a filtering router or the TCP wrapper.

Other services should receive the same treatment. There's generally no need to run *ftpd* on most machines; turn it off on the others. CERT publishes a list of dangerous ports (see Section A.5.1); block these at the front door if you can.

The filtering configuration should not be considered static. If someone is traveling off-premises, any necessary services can be enabled for the duration. Thus, access to X11 can be turned on, but only for a given set of machines. Be very careful, though, to keep track of such "temporary" changes; a formal mechanism for tracking and undoing them is useful.

Finally, make sure that you know which machines are trusted. Your subnet may be secure, but if you permit *rlogin*s from the outside, your machines may be compromised via an indirect path. Again, packet filters, wrappers, or even an enforceable prohibition on the use of `.rhosts` files should be used.