3

Firewall Gateways

fire wall *noun*: A fireproof wall used as a barrier to prevent the spread of a fire.

—AMERICAN HERITAGE DICTIONARY

3.1 Firewall Philosophy

Up to this point, we have used the words "firewall" and "gateway" rather casually. We will now be more precise. A *firewall*, in general, consists of several different components (Figure 3.1). The "filters" (sometimes called "screens") block transmission of certain classes of traffic. A *gateway* is a machine or a set of machines that provides relay services to compensate for the effects of the filter. The network inhabited by the gateway is often called the *demilitarized zone* (*DMZ*). A gateway in the DMZ is sometimes assisted by an *internal gateway*. Typically, the two gateways will have more open communication through the inside filter than the outside gateway has to other internal hosts. Either filter, or for that matter the gateway itself, may be omitted; the details will vary from firewall to firewall. In general, the outside filter can be used to protect the gateway from attack, while the inside filter is used to guard against the consequences of a compromised gateway. Either or both filters can protect the internal network from assaults. An exposed gateway machine is often called a *bastion host*.

We classify firewalls into three main categories: *packet filtering*, *circuit gateway*s, and *application gateway*s. Commonly, more than one of these is used at the same time. As noted earlier, mail is often routed through a gateway even when no security firewall is used.

3.1.1 Costs

Firewalls are not free. Costs include:

- hardware purchase,

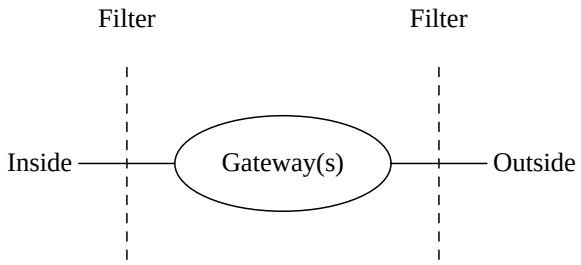


Figure 3.1: Schematic of a firewall.

- hardware maintenance,
- software development or purchase,
- software update costs,
- administrative setup and training,
- ongoing administration and trouble-shooting,
- lost business or inconvenience from a broken gateway or blocked services, and
- the loss of some services or convenience that an open connection would supply.

These must be weighed against the costs of not having a firewall:

- the effort spent in dealing with break-ins (i.e., the costs of a gateway failure), including lost business, and
- legal and other costs of sponsoring hacker activity.

These cost considerations vary greatly depending on the kind of site one is protecting. Modern computers are fairly cheap, but a university might not justify such a purchase for a dedicated gateway machine. For them, student labor can keep the administrative costs low. Universities often believe that open access to the Internet is part of creating an open community. Of course, we have found that 90% of our hacker probes come from such open communities (Chapter 11). Universities do have administrative computers that definitely need protection. Some students have the time and motivation to seek out the payroll, alumni, and especially the grades databases.

A large company would consider $30,000 worth of hardware cheap insurance if management gains some assurance that the company secrets will not leak. To the boss the lack of certain services is regrettable but necessary. Corporate lawyers are quick to worry about potential liability for harboring hackers, though we are unaware of any such lawsuits. More subtly, the management of a corporation might be liable for neglecting their fiduciary duties to the shareholders. For whatever reason, in our experience companies from which hacker attacks originate have been very quick to solve the problem.

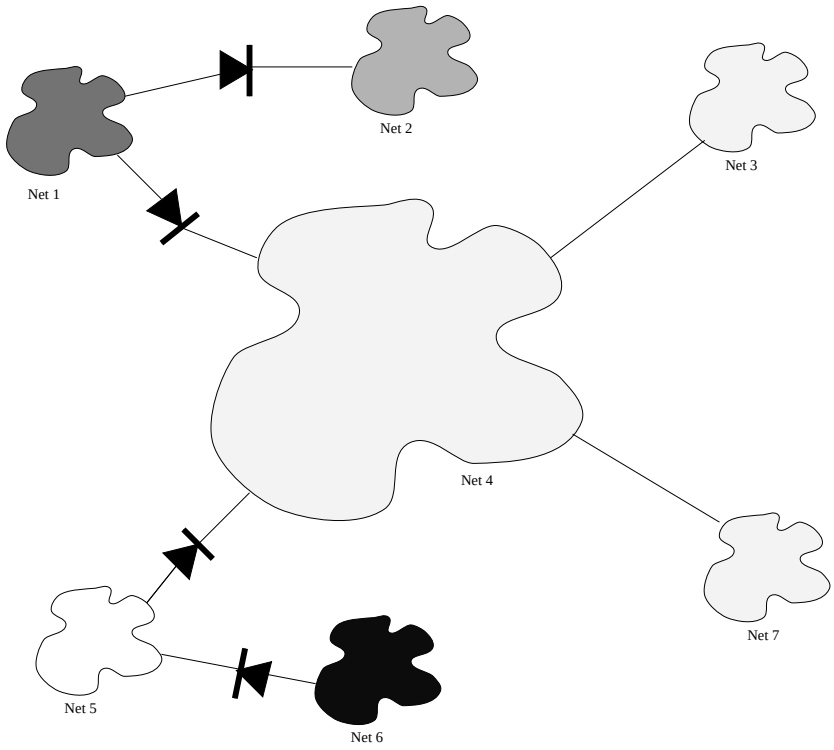


Figure 3.2: Positioning firewalls.

3.2 Situating Firewalls

Traditionally, firewalls are placed between an organization and the outside world. But a large organization may need internal firewalls as well to isolate *security domains* (also known as *administrative domains*). A security domain is a set of machines under common administrative control, with a common security policy and security level.

Consider the network shown in Figure 3.2. The different security domains are indicated by different shadings. Firewalls, shown by a diode symbol (⊣▶⊢), should be positioned at the boundaries between security domains. The arrow in the diode points to the bad guys. In this case, we see that NET 1 does not trust any other network, not even NET 2, even though the latter appears to be an internal net, since it is attached directly to NET 1.

There are many good reasons to erect internal firewalls. In many large companies, most employees are not (or should not be) privy to all information. In other companies, the cash business (like the factory, or a phone company's telephone switches) needs to be accessible to developers or support personnel, but not to the general corporate population. Even authorized

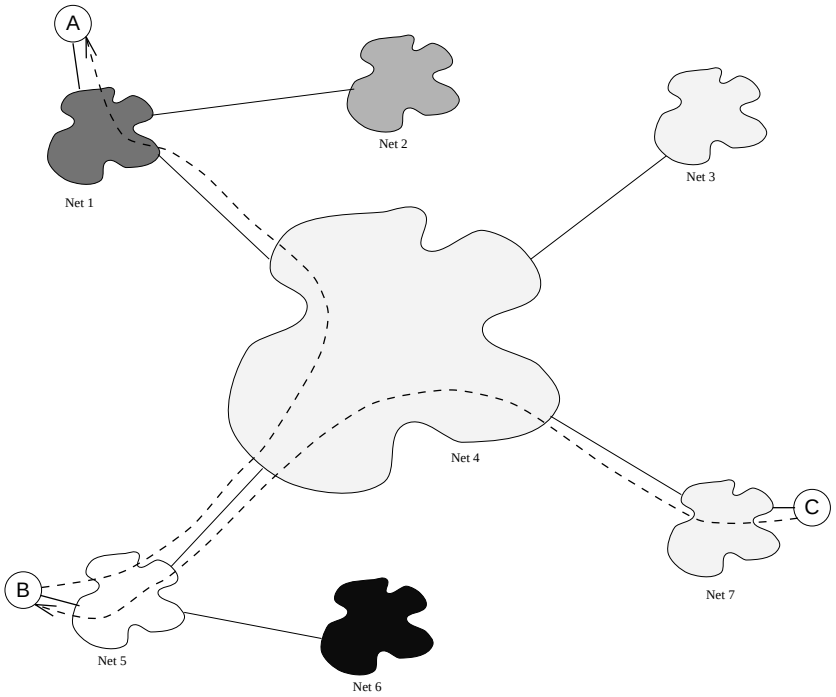


Figure 3.3: An example of transitive trust.

users should pass through a security gateway when crossing the firewall; otherwise, if their home machines, which live outside of the firewall, are compromised, the sensitive equipment on the inside could be next. The firewall controls the access and the trust in a carefully predictable way.

Transitive trust may also be an issue. In Figure 3.3 suppose that machine A, in full accord with local security policy, decides to extend trust to machine B. Similarly, machine B decides to trust machine C, again in accordance with its own policies. The result, though, is that machine A is now trusting machine C, whether it wants to or not, and whether it knows it or not. A firewall will prevent this. The diodes from Figure 3.2 would prevent machine A from trusting machine B, or machine B from trusting machine C. Again, trust can be controlled through a firewall. But machine C could, if it wished, trust machine B.

3.3 Packet-Filtering Gateways

Packet filters can provide a cheap and useful level of gateway security. Used by themselves, they are cheap: the filtering abilities come with the router software. Since you probably need a router

to connect to the Internet in the first place, there is no extra charge. Even if the router belongs to your network service provider, you'll probably find that they'll install any filters you wish.

Packet filters work by dropping packets based on their source or destination addresses or ports. In general, no context is kept; decisions are made only from the contents of the current packet. Depending on the type of router, filtering may be done at input time, at output time, or both. The administrator makes a list of the acceptable machines and services and a stoplist of unacceptable machines or services. It is easy to permit or deny access at the host or network level with a packet filter. For example, one can permit any IP access between host A and B, or deny any access to B from any machine but A.

Most security policies require finer control than this: they need to define access to specific services for hosts that are otherwise untrusted. For example, one might want to allow any host to connect to machine A, but only to send or receive mail. Other services may or may not be permitted. Packet filtering allows some control at this level, but it is a dangerous and error-prone process. To do it right, one needs intimate knowledge of TCP and UDP port utilization on a number of operating systems.

> *This is one of the reasons we do not like packet filters very much: as Chapman [1992] has shown, if you get these tables wrong you may inadvertently let in the Bad Guys.*

Even with a perfectly implemented filter, some compromises can be dangerous. We discuss these later.

Configuring a packet filter is a three-step process. First, of course, one must know what should and should not be permitted. That is, one must have a security policy, as explained in Section 1.2. Next, the allowable types of packets must be specified formally, in terms of logical expressions on packet fields. Finally—and this can be remarkably difficult—the expressions must be rewritten in whatever syntax your vendor supports.

An example is helpful. Suppose that one part of your security policy was to allow inbound mail (SMTP, port 25), but only to your gateway machine. However, mail from some particular site SPIGOT is to be blocked, because of their penchant for trying to mail several gigabytes of data at a time. A filter that implemented such a ruleset might look like this.

action	ourhost	port	theirhost	port	comment
block	*	*	SPIGOT	*	*we don't trust these people*
allow	OUR-GW	25	*	*	*connection to our SMTP port*

The rules are applied in order from top to bottom. Packets not explicitly allowed by a filter rule are rejected. That is, every ruleset is followed by an implicit rule reading like this.

action	ourhost	port	theirhost	port	comment
block	*	*	*	*	*default*

This fits with our general philosophy: all that is not expressly permitted is prohibited.

Note carefully the distinction between the first ruleset, and the one following, which is intended to implement the policy "any inside host can send mail to the outside."

action	ourhost	port	theirhost	port	comment
allow	*	*	*	25	*connection to their SMTP port*

The call may come from any port on an inside machine, but will be directed to port 25 on the outside. This ruleset seems simple and obvious. It is also wrong.

The problem is that the restriction we have defined is based solely on the outside host's port number. While port 25 is indeed the normal mail port, there is no way we can control that on a foreign host. An enemy can access any internal machine and port by originating his call from port 25 on the outside machine.

A better rule would be to permit *outgoing* calls to port 25. That is, we want to permit our hosts to make calls to someone else's port 25, so that we know what's going on: mail delivery. An incoming call *from* port 25 implements some service of the caller's choosing. Fortunately, the distinction between incoming and outgoing calls can be made in a simple packet filter if we expand our notation a bit.

A TCP conversation consists of packets flowing in two directions. Even if all of the data is flowing one way, acknowledgment packets and control packets must flow the other way. We can accomplish what we want by paying attention to the direction of the packet, and by looking at some of the control fields. In particular, an initial open request packet in TCP does not have the `ACK` bit set in the header; all other TCP packets do. [Strictly speaking, that is not true. Some packets will have just the reset (`RST`) bit set. This is an uncommon case, which we do not discuss further, except to note that one should generally allow naked `RST` packets through one's filters.] Thus, packets with `ACK` set are part of an ongoing conversation; packets without it represent connection establishment messages, which we will permit only from internal hosts. The idea is that an outsider cannot initiate a connection, but can continue one. One must believe that an inside kernel will reject a continuation packet for a TCP session that has not been initiated. To date, this is a fair assumption. Thus, we can write our ruleset as follows, keying our rules by the source and destination fields, rather than the more nebulous "OURHOST" and "THEIRHOST":

action	src	port	dest	port	flags	comment
allow	{our hosts}	*	*	25		*our packets to their SMTP port*
allow	*	25	*	*	`ACK`	*their replies*

The notation "{our hosts}" describes a set of machines, any one of which is eligible. In a real packet filter, you could either list the machines explicitly, or you could specify a group of machines, probably by the network number portion of the IP address.

3.3.1 Handling IP Fragments

The existence of IP fragmentation makes life difficult for packet filters. Except for the first one, fragments do not contain port numbers; there is thus little information on which to base a filtering decision. The proper response depends on the goals you have chosen for your firewall.

If the main threat is penetration attempts from the outside, fragments can be passed without further ado. The initial fragment will have the port number information and can be processed

appropriately. If it is rejected, the packet will be incomplete, and the remaining fragments will eventually be discarded by the destination host.

If, however, information leakage is a significant concern, fragments must be discarded. Nothing prevents someone intent on exporting data from building bogus noninitial fragments and converting them back to proper packets on some outside machine.

You can do better if your filter keeps some context. Mogul's *screend* [Mogul, 1989] caches the disposition and salient portion of the header for any initial fragment, and subsequent pieces of the same packet will share its fate.

3.3.2 Filtering FTP Sessions

At least three major services are not handled well by packet filters: FTP, X11, and the DNS. The problems with the DNS concern the sensitivity of the information itself, as discussed in Section 2.3; a possible solution is discussed in Section 3.3.4. But the issues surrounding the first two are clear-cut: normal operation demands use of incoming calls. (The *rsh* command uses an incoming call as well, for `stderr`. However, it is rarely used through firewalls, although there is no inherent reason why it couldn't be.)

For FTP, files are transferred via a secondary connection. If the control channel to a server on THEIRHOST uses the connection

$$\langle \textit{ourhost, ourport, theirhost, 21} \rangle,$$

file transfers will occur on

$$\langle \textit{ourhost, ourport, theirhost, 20} \rangle$$

by default. Furthermore, the server must initiate the file transfer call. We thus have the problem we saw earlier, but without the ability to screen based on the direction of the call.

One idea is to use the range of *ourport* to make filtering decisions. Most servers, and hence most attack targets, live on low-numbered ports; most outgoing calls tend to use higher numbered ports, typically above 1023. Thus, a sample ruleset might be

action	src	port	dest	port	flags	comment
allow	{our hosts}	*	*	*		*our outgoing calls*
allow	*	*	*	*	`ACK`	*replies to our calls*
allow	*	*	*	$\geq$ 1024		*traffic to nonservers*

That is, packets are passed under one of three circumstances:

1. They originated from one of our machines,
2. They are reply packets to a connection initiated by one of our machines,
3. They are destined for a high-numbered port on our machines.

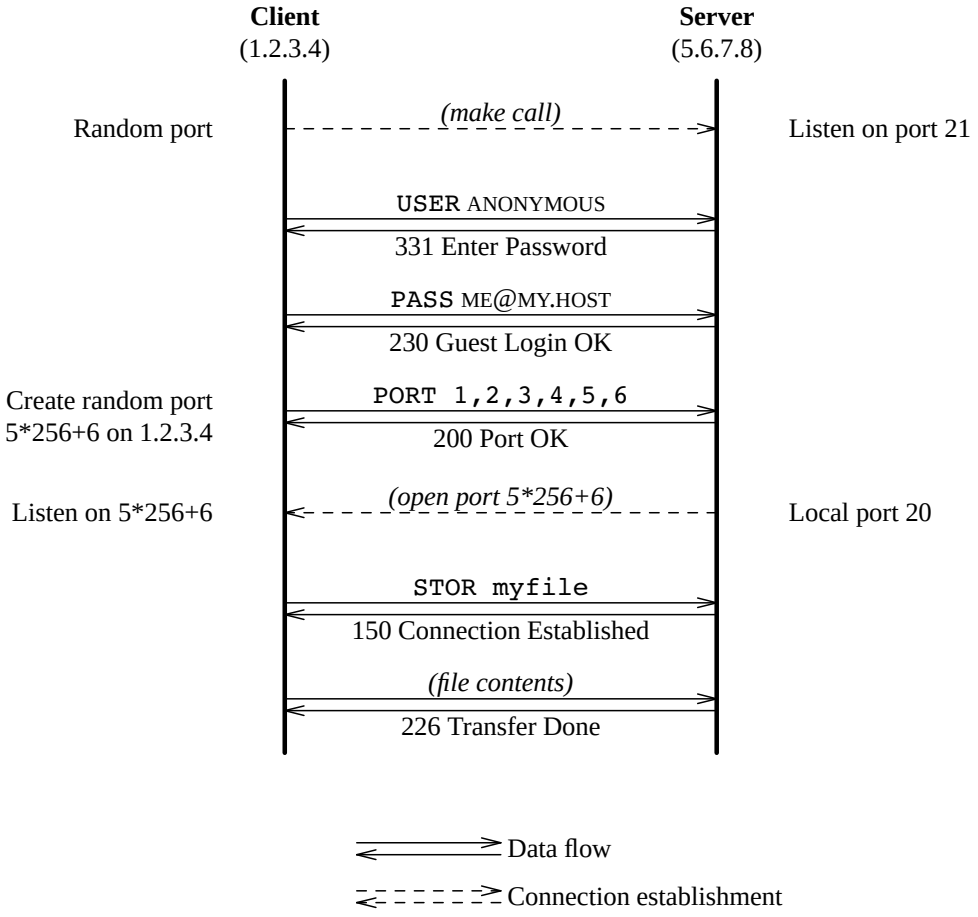


Figure 3.4: Using FTP normally.

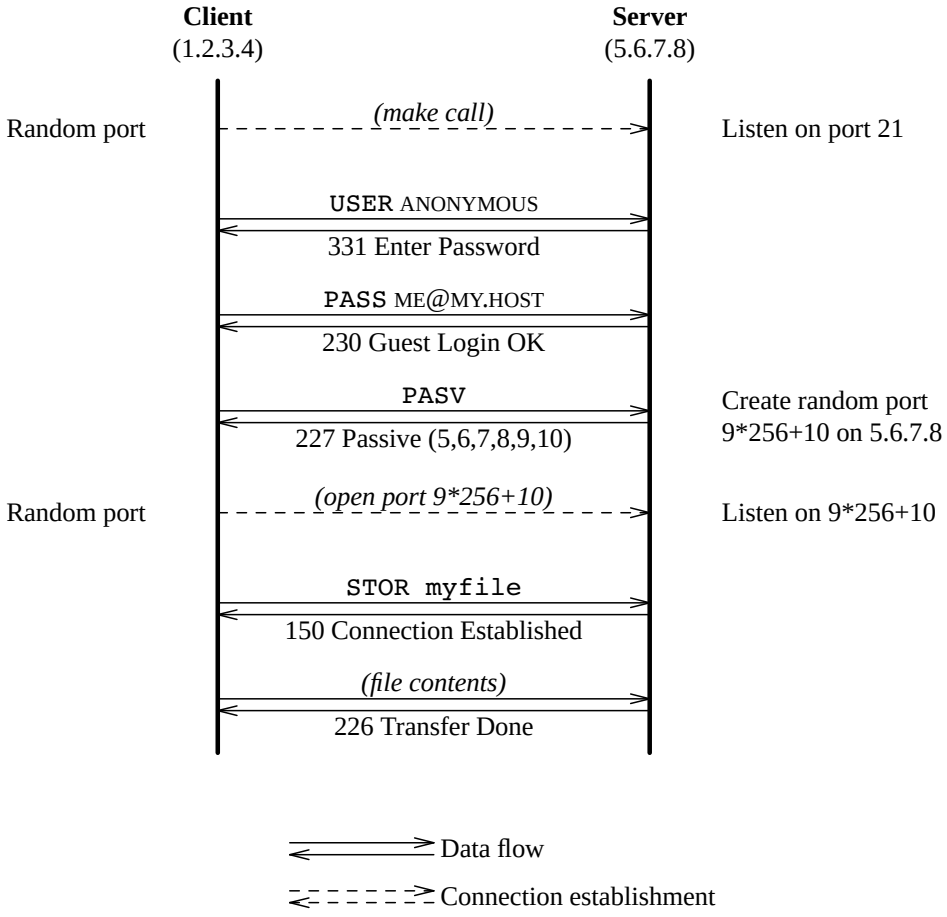


Figure 3.5: Using FTP with the `PASV` command.

Strictly speaking, the last two rules apply to all packets, not just packets originating from outside. But any packets from the inside would be accepted by the first rule, and would not be examined by the later rules.

Unfortunately, this ruleset does not accomplish what we really want, which is to block incoming calls to our servers. We said "most servers" live on low-numbered ports, not "all." A number of tempting targets, especially X11, inhabit high-numbered ports. Presumably, one could filter out known dangerous ports; unfortunately, new ones could be added without notice. Thus, a cautious stance dictates that this heuristic not be adopted. Under certain circumstances, a bypass is available if you have the source code to the FTP client programs. You can modify the programs to issue a `PASV` command to the server, directing it to do a passive open, and thus permitting an *outgoing* call through the firewall for the data channel. The difference is shown in Figures 3.4 and 3.5. In the former—the default—the client (1.2.3.4) picks a random port and announces it via the `PORT` command; the server opens a connection to it. In the latter, the modified client program sends a `PASV` command; in turn, the server (5.6.7.8) generates a random port and asks the client to initiate the connection.

This variant is not without its problems. The data channel, though an outgoing call, is to a random port. Such calls are generally barred by sites that wish to restrict outbound data flow. You also have the obvious problem of distributing modified clients to all inside machines. Also, not all servers understand the `PASV` command, even though they should. The issues are discussed further in [Bellovin, 1994].

3.3.3 Filtering X Window sessions

The problem with X11 is similar to FTP in one respect: proper use requires an incoming call. That is, the user's display—screen, keyboard, and mouse—is a server; X11 applications connect to it via TCP. If the applications are to be run on outside hosts, the connection to the server involves a call made from the outside, which typical rulesets block. This is especially annoying, since it represents the category of application—an internal user wishing to use external facilities—that is typically permitted and desirable.

Unauthorized X11 connections are a considerable threat. Intruders can dump data from screens, monitor keystrokes, and—under certain circumstances—generate bogus keyboard input. At a recent conference someone caused all of the public X11 terminals to display advertising for his company. He could have done worse things, and he could have done them to many X11 terminals around the Internet.

To some extent, the threat posed by X11 can be handled by cooperation from the user community and proper configuration of the X11 servers. Conversely, small errors can cause very serious consequences. As usual, we prefer to err on the side of caution: we recommend blocking any inbound calls to port numbers 6000–6100, at the very least. (If you have more than 100 X11 servers running on any single host, you should protect the whole range plus a safety margin, of course.)

There is one possible side effect: if you block *all* traffic to those ports, rather than just incoming calls, you run the risk of upsetting random client programs that just happened to be assigned port numbers in the forbidden range.

```
bar.com.        IN      SOA     foo.bar.com.  hostmaster.foo.bar.com. (
                        9404011 ;serial
                        3600    ;refresh
                        900     ;retry
                        604800  ;expire
                        86400 ) ;minim

bar.com.        IN      NS      foo.bar.com.
bar.com.        IN      NS      x.trusted.edu.
foo.bar.com.    IN      A       200.2.3.4
x.trusted.edu.  IN      A       5.6.7.8

foo.bar.com.    IN      MX      0 foo.bar.com.
*.bar.com.      IN      MX      0 foo.bar.com.
bar.com.        IN      MX      0 foo.bar.com.

ftp.bar.com.    IN      CNAME   foo.bar.com.
```

Figure 3.6: A minimal DNS zone. The inverse mapping tree is similarly small. Note the use of an alias for the FTP server. The secondary server (X.TRUSTED.EDU) is a sensitive site; any hacker who corrupted it, perhaps via a site that it trusts, could capture much of your inbound mail and intercept many incoming *telnet* calls.

One more property of X11 bears mentioning. On occasion, a legitimate inside user will need to run an application on an internal machine from an external server. This represents an outgoing call through the firewall, which is normally not a problem. However, the characteristics of the remote X11 server make this somewhat more dangerous. First, there is a nontrivial risk that the server will be penetrated. But that would be a risk even if the user simply used *telnet* or some such means to call in. The incremental risk is comparatively low, although the truly cautious may wish to ban both activities. More seriously, with X11, window managers are simply applications that use special primitives. They can live anywhere any other application can live. If the user runs a window manager on the inside of the firewall, an attacker can generate synthetic mouse movements to ask the window manager to create new processes on the inside. These could be invisible to the terminal user.

3.3.4 Taming the DNS

Dealing with the DNS is one of the more difficult problems in setting up a firewall. It is utterly vital that the gateway machine use it, but it poses many risks.

What tack you take depends on the nature of your firewall. If you run a circuit or application gateway, there is no need to use the external DNS internally. The information you advertise to the outside world can be minimal (see Figure 3.6). It lists the name server machines themselves (FOO.BAR.COM and X.TRUSTED.EDU), the FTP and mail relay machine (FOO.BAR.COM again), and it says that all mail for any host in the BAR.COM domain should be routed to the relay.

Of course, the inside machines can use the DNS if you choose; this depends on the number of hosts and system administrators you have. If you do, you must run an isolated internal DNS with its own pseudo-root. We do that, but we are careful to follow all of the necessary conventions for the "real" DNS.

Life is much more difficult for sites that use packet filters. As noted, one does *not* want to expose the DNS to curious minds and fingers; however, inside hosts need to use the DNS to reach outside sites. In some messages to the Firewalls mailing list, Chapman has described a scheme that works today because of the way most UNIX system name servers happen to be implemented. But it is not guaranteed to work with all systems.

His approach (Figure 3.7) is to run name servers for the domain on both the gateway machine and on some inside machine. The latter has the real information—the gateway's name server has the sort of minimal file shown in Figure 3.6. Thus, outside machines have no access to the sensitive internal information.

The tricky parts are

1. permitting the gateway itself to resolve internal names for, say, mail delivery,
2. permitting inside machines to resolve external names, and
3. providing a way for the necessary UDP packets to cross the firewall.

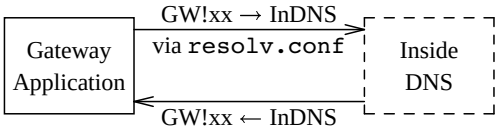
The first part is handled by creating a `/etc/resolv.conf` file on the gateway that points to the internal DNS server. That file tells *application programs* on the gateway, but not the name server itself, where to go to resolve queries. Thus, whenever, say, *mail* wants to find an IP address, it will ask the inside server.

Name server processes pay no attention to `/etc/resolv.conf` files. They simply use the tree-structured name space and their knowledge of the root name servers to process all requests. Queries for names they do not know are thus properly resolved.

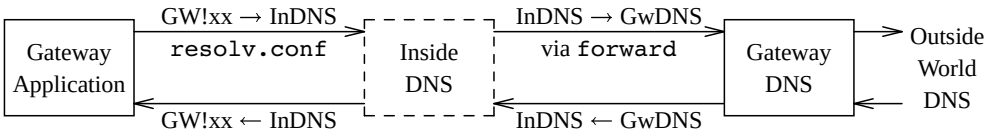
The second problem involves queries for external names sent to the internal name server. Of course, this server doesn't know about outside machines. Rather than talking to the real servers directly (we cannot permit that, since we can't get the replies through the firewall safely), the inside server has a `forward` line pointing to the gateway in its configuration file. This line denotes which server should be queried for any names not known locally. Thus, if asked about an inside machine, it responds directly; if asked about an outside machine, it passes the query to the gateway's name server.

Note the curious path taken by a request for an outside name by a process running on the gateway machine. It first goes to the inside server, which *can't* know the answer unless it's cached. It then hops back across the firewall to the outside machine's own server, and thence eventually to the distant DNS server that really knows the answer. The reply travels the same twisty path.

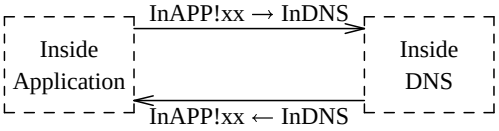
The reason that the inside and outside servers can talk through the packet filter is that both servers use the official DNS UDP port (53) as the source port number when forwarding queries. There is no requirement that they do so: it is simply a feature of the implementation. But it does permit a safe filter rule that accepts packets from the outside machine if they are destined for the inside server's port 53. (But make sure that your router's filter is configured to prevent source address forgery on such packets.) This solves the third problem.



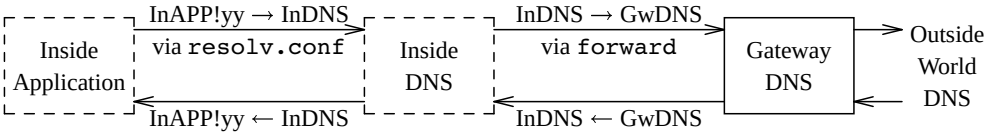
(a) Gateway application calling inside machine.



(b) Gateway application calling outside machine.



(c) Inside application calling inside machine.



(d) Inside application calling outside machine.

Figure 3.7: Passing DNS through a packet filter. The packet filter separates the gateway machine GW from the inside machines; the latter are always shown as dashed boxes. Note that all incoming packets through the firewall—that is, all arrows from solid boxes to dashed ones—are from GW and to the inside DNS server INDNS, which lives on port 53. The query always starts out in the left-most box; in scenario (b), the query goes back out through the firewall, as noted in the text.

One "ı" has been left undotted. If an inside machine opens a connection to some external site, that site will probably want to look up its host name. But the gateway's DNS server does not have that information, and this sort of failure will cause many sites to reject the connection. For example, a number of FTP sites require that the caller's IP address be listed in the DNS. Chapman suggests using a wild card `PTR` record:

```
*.3.2.127.in-addr.arpa.    IN   PTR    UNKNOWN.bar.com.
```

which will a least offer some answer to the query. But if the external site performs a DNS cross-check, as described in Section 2.3, it will fail; again, many outside sites will reject connections if this occurs. UNKNOWN.BAR.COM has no IP addresses corresponding to the actual inside machine's address. To deal with that, a more complete fiction is necessary. One suggestion we've heard is to return a special-format host name for any address in your domain:

```
42.3.2.127.in-addr.arpa.  IN  PTR  pseudo_127_2_3_42.bar.com.
```

When a query is made for an `A` record for names of this form, the appropriate record can be synthesized.

3.3.5 Protocols without Fixed Addresses

Some services are problematic for packet filters because they can involve random port numbers. On occasion the situation is even worse: a number of services *always* use random port numbers, and rely on a separate server to supply the current contact information.

Two examples of this are the *tcpmux* protocol [Lottor, 1988] and the *portmapper* [Sun Microsystems, 1990] used by SunOS for RPC [Sun Microsystems, 1988]. In both cases, client programs contact the mapping program rather than the application. The *portmapper* also processes registration requests from applications, informing it of their current port numbers. On the other hand, *tcpmux* will invoke the application directly, passing it the open connection.

This difference gives rise to different filter-based protection mechanisms. With *tcpmux*, one can block access to either all such services, or none, simply by controlling access to the *tcpmux* port. With the *portmapper*, each service has its own port number. While one can deny easy access to them by filtering out *portmapper* requests, an intruder can bypass the portmapper and simply sweep the port number space looking for interesting applications. We have seen evidence of this happening. The only cure is to block access to all possible port numbers used by RPC-based servers—and there's no easy way to know what that range is.

3.3.6 Filter Placement

Packet filtering can occur in several places. Figure 3.8 shows a typical router. Packets can be examined at point (a), point (b), or both. Furthermore, filters can be applied to incoming packets, outgoing packets, or both. Not all routers permit all of these possibilities, and some have a few more possibilities; obviously, this affects the style of filters used.

Filtering packets on the way out of the router may increase efficiency, since finding and applying the filter rule can often be combined with the routing table lookup. On the other hand,

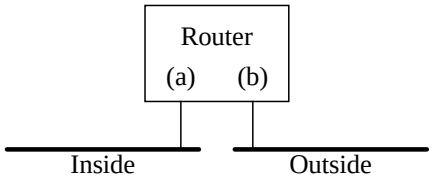


Figure 3.8: A typical firewall router.

some information has been discarded, like knowledge of the physical wire on which the packet arrived. This is especially important in preventing address-spoofing attacks (see Section 3.3.7). Filtering on input can protect the router itself from attack.

As a general policy, filter out the offending packets as soon as possible. Do not rely on deleting TCP's responses; attacks are possible even if the attacker never sees an answer [Bellovin, 1989]. Thus, one should write our first ruleset as

action	src	port	dest	port	comment
block	SPIGOT	*	*	*	*we don't trust these people*
allow	*	*	OUR-GW	25	*connection to our SMTP port*
allow	OUR-GW	25	*	*	*our reply packets*

rather than as the following filter on our responses:

action	src	port	dest	port	comment
block	*	*	SPIGOT	*	*this rule is subtly different*
allow	*	*	OUR-GW	25	
allow	OUR-GW	25	*	*	

Note that for this example, the rulesets would be the same for points (a) and (b).

Some routers will only filter on the destination port, rather than on both source and destination. The idea is that since all TCP conversations involve bidirectional traffic, every message *to* a given port will generate an acknowledgment message *from* that port. However, since components of the two rules are not tied together, dangerous interactions can occur. The following example, taken from [Chapman, 1992], illustrates this nicely.

Suppose that you wish to permit incoming and outgoing mail but nothing else. A mail connection is characterized by a destination port number of 25, and a source port in the high-numbered range. At point (a), we would use the following filter on output packets, i.e., on packets leaving the router:

action	dest	port	comment
allow	*	25	*Incoming mail*
allow	*	$\geq$ 1024	*Outgoing mail response packets*
block	*	*	*Nothing else*

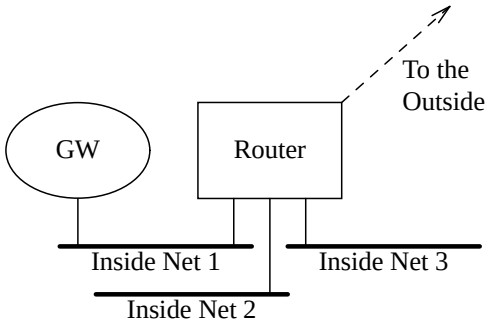


Figure 3.9: A firewall router with multiple internal networks.

That is, we allow traffic if it is either to our mailer daemon or if it appears to be their responses to outgoing messages from our mailer. The same ruleset is used at point (b) to permit our calls to their mailer and our responses to their messages. Consider, though, what would happen to a conversation between two high-numbered ports. Both filters would permit the packet to pass, since the condition *destport* $\geq$ 1024 is satisfied. Such a connection may not be evil, but it was not what was intended by the administrator.

Strictly speaking, there is a third choice for filter placement: without regard to interface at all. The *screend* filter behaves this way. All decisions are made solely on the basis of the addresses in the packet; there is thus no protection against address-spoofing.

A totally different set of problems can arise if the firewall applies your filter rules in some other order than the one you specify (and some do). Writing rulesets is hard enough; to have them rearranged without warning is unacceptable. Chapman gives examples of the unintended consequences that can result from such behavior [Chapman, 1992]. Fortunately, some router manufacturers are starting to follow his advice on how to improve things.

3.3.7 Network Topology and Address-Spoofing

For reasons of economy, it is sometimes desirable to use a single router both as a firewall and to route internal-to-internal traffic. Consider the network shown in Figure 3.9. There are four networks, one external and three internal. One is inhabited solely by a gateway machine GW. The intended policies are as follows.

- Very limited connections are permitted through the router between GW and the outside world.
- Very limited, but possibly different, connections are permitted between GW and anything on NET 2 or NET 3. This is protection against GW being compromised.
- Anything can pass between NET 2 or NET 3.
- Outgoing calls only are allowed between NET 2 or NET 3 and the external link.

What sorts of filter rules should be specified? This situation is very difficult if only output filtering is done. First, a rule permitting open access to NET 2 must rely on a source address belonging to NET 3. Second, nothing prevents an attacker from sending in packets from the outside that claim to be from an internal machine. Vital information—that legitimate NET 3 packets can only arrive via one particular wire—has been ignored.

Address-spoofing attacks like this are difficult to mount, but are by no means out of the question. Simple-minded attacks using IP source-routing are almost foolproof, unless your firewall filters out these packets. But there are more sophisticated attacks as well. A number of these are described in [Bellovin, 1989]. Detecting them is virtually impossible unless source-address filtering and logging are done.

Such measures do not eliminate all possible attacks via address-spoofing. An attacker can still impersonate a host that is trusted but not on an internal network. One should not trust hosts outside of one's administrative control.

Assume, then, that filtering takes place on input, and that we wish to allow any outgoing call, but permit incoming calls only for mail, and only to our gateway GW. The ruleset for the external interface should read:

action	src	port	dest	port	flags	comment
block	{NET 1}	*	*	*		*block forgeries*
block	{NET 2}	*	*	*		
block	{NET 3}	*	*	*		
allow	*	*	GW	25		*legal calls to us*
allow	*	*	{NET 2}	*	`ACK`	*replies to our calls*
allow	*	*	{NET 3}	*	`ACK`	

That is, prevent address forgery, and permit incoming packets if they are to the mailer on the gateway machine, or if they are part of an ongoing conversation initiated by any internal host at all. Anything else will be rejected.

Note one detail: our rule specifies the destination host GW, rather than the more general "something on NET 1". If there is only one gateway machine, there is no reason to permit open access to that network. If several hosts collectively formed the gateway, one might opt for simplicity, rather than this slightly tighter security; on the other hand, if the different machines served different roles, one might prefer to limit the connectivity to each gateway host to the services it was intended to handle.

The ruleset on the router's interface to NET 1 should be only slightly less restrictive than this one. Choices here depend on one's stance. It certainly makes sense to bar unrestricted internal calls, even from the gateway machine. Some would opt for mail delivery only. We opt for more caution; our gateway machine will speak directly only to other machines running the *upas* mailer, since we do not trust *sendmail*. One such machine is an internal gateway. The truly paranoid do not permit even this. Rather, a relay machine will call out to GW to pick up any waiting mail. At most, a notification is sent by GW to the relay machine. The intent here is to guard against common-mode failures: if a gateway running *upas* can be subverted that way, internal hosts running the same software can (probably) be compromised in the same fashion.

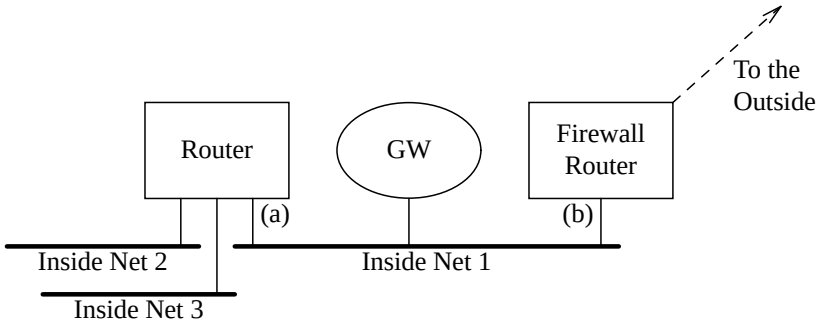


Figure 3.10: A firewall with output-filtering routers.

Our version of the ruleset for the NET 1 interface reads like this:

action	src	port	dest	port	flags	comment
allow	GW	*	{*partners*}	25		*mail relay*
allow	GW	*	{NET 2}	*	`ACK`	*replies to inside calls*
allow	GW	*	{NET 3}	*	`ACK`	
block	GW	*	{NET 2}	*		*stop other calls from* GW
block	GW	*	{NET 3}	*		
allow	GW	*	*	*		*let* GW *call the world*

Again, we prevent spoofing, because the rules all specify GW; only the gateway machine is supposed to be on that net, so nothing else should be permitted to send packets.

If we are using routers that support only output filtering, the recommended topology looks very much like our schematic diagram (Figure 3.1). We now need two routers to accomplish the tasks that one router was able to do earlier (Figure 3.10). At point (a) we use the ruleset that protects against compromised gateways; at point (b) we use the ruleset that guards against address forgery and restricts access to only the gateway machine. We do not have to change the rules even slightly. Assuming that packets generated by the router itself are not filtered, in a two-port router an input filter on one port is exactly equivalent to an output filter on the other port.

Input filters do permit the router to deflect packets aimed at it. Consider the following rule:

action	src	port	dest	port	flags	comment
block	*	*	ROUTER	*		*prevent router access*

This rejects all nonbroadcast packets destined for the firewall router itself. This rule is probably too strong. One almost certainly needs to permit incoming routing messages. It may also be useful to enable responses to various diagnostic messages that can be sent from the router. Our general rule holds, though: if you do not need it, eliminate it.

One more point bears mentioning if you are using routers that do not provide input filters. The external link on a firewall router is often a simple serial line to a network provider's router. If

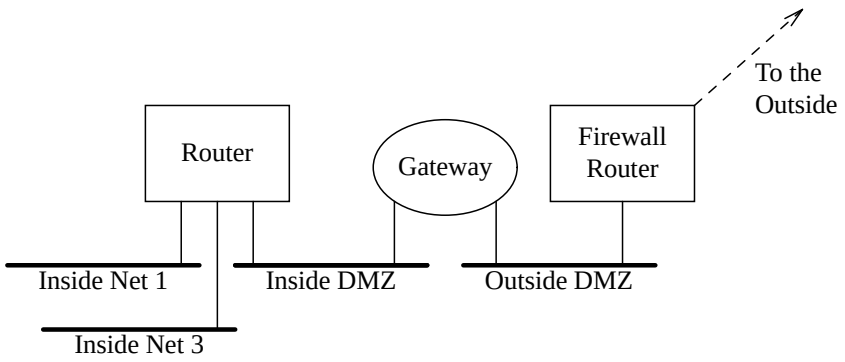


Figure 3.11: A "belt-and-suspenders" firewall.

you are willing to trust the provider, filtering can be done on the output side of that router, thus permitting use of the topology shown in Figure 3.9. But caution is needed: the provider's router probably serves many customers, and hence is subject to more frequent configuration changes. The chances of an accident are correspondingly higher. Furthermore, the usefulness of the network provider's router relies on the line being a simple point-to-point link; if you are connected via a multipoint technology, such as X.25, frame relay, or ATM, it may not work.

A rather paranoid configuration, for an application or circuit gateway, is shown in Figure 3.11. In this variant, which we call *belt-and-suspenders*, the gateway machine sits on two different networks, between the two filtering routers. It is an ordinary gateway, except in one respect: it *must* be configured not to forward packets, either implicitly or via IP source routing. This can be harder than it seems; some kernels, though configured not to forward packets, will still do so if source routing is used. If you have access to kernel source, we suggest that you rip out the packet-forwarding code. The outside router should be configured to allow access only to desired services on the gateway host; additionally, it should reject any packet whose apparent source address belongs to an inside machine. In turn, the gateway machine should use its own address filtering to protect restricted services, such as application or circuit relays. The inside filter should permit access only to the hosts and ports that the gateway is allowed to contact.

The theory behind this configuration is simple: the attacker must penetrate not just the packet filters on the router, but also the gateway machine itself. Furthermore, even if that should occur, the second filter will protect most inside machines from the now-subverted gateway.

3.3.8 Packet Filters and UDP

Filtering TCP circuits is difficult. Filtering UDP packets while still retaining desired functionality is all but impossible. The reason lies in the essential difference between TCP and UDP: the former is a virtual circuit protocol, and as such has retained context; the latter is a datagram protocol, where each message is independent. As we saw earlier, filtering TCP requires reliance on the `ACK` bit, in order to distinguish between incoming calls and return packets

from an outgoing call. But UDP has no such indicator: we are forced to rely on the source port number, which is subject to forgery.

An example will illustrate the problem. Suppose an internal host wishes to query the UDP *echo* server on some outside machine. The originating packet would carry the address

$$\langle \textit{localhost, localport, remotehost, 7} \rangle,$$

where *localport* is in the high-numbered range. But the reply would be

$$\langle \textit{remotehost, 7, localhost, localport} \rangle,$$

and the router would have no idea that *localport* was really a safe destination. An incoming packet

$$\langle \textit{remotehost, 7, localhost, 2049} \rangle,$$

is probably an attempt to subvert our NFS server; and, while we could list the known dangerous destinations, we do not know what new targets will be added next week by a system administrator in the remote corners of our network. Worse yet, the RPC-based services use dynamic port numbers, sometimes in the high-numbered range. As with TCP, indirectly named services are not amenable to protection by packet filters.

A conservative stance therefore dictates that we ban virtually all *outgoing* UDP calls. It is not that the requests themselves are dangerous; rather, it is that we cannot trust the responses. The only exceptions are those protocols where there is a peer-to-peer relationship. A good example is NTP, the Network Time Protocol. In normal operation, messages are both from and to port 123. It is thus easy to admit replies, because they are to a fixed port number, rather than to an anonymous high-numbered port. But one use of NTP—setting the clock when rebooting—will not work, because the client program will not use port 123. (Of course, a booting computer probably shouldn't ask an outsider for the time.)

3.3.9 Filtering Other Protocols

Other protocols are layered on top of IP as well; depending on your environment, these may need to be filtered also. Of particular import is ICMP, the Internet Control Message Protocol. There have been instances of hackers abusing it for denial-of-service attacks. On the other hand, filtering out ICMP denies one useful information. At the very least, internal management hosts should be allowed to receive such messages so that they can perform network diagnostic functions. For example, *traceroute* relies on the receipt of `Time Exceeded` and `Port Invalid` packets.

Some routers can distinguish between "safe" and "unsafe" ICMP messages, or permit the filter to specify the message types explicitly. This lets more of your machines send and respond to things like *ping* requests. On the other hand, it lets an outsider map your network.

Most of the other higher level protocols are not important in most environments. Still, if you do use others, the same care needs to be taken as for TCP and UDP. One such protocol of growing importance is the IP-over-IP protocol used by the MBone; if you are not careful, it can be used to bypass your firewall. Router filter lists should also be configured to reject all unneeded protocols.

When Routes Leak

Once upon a time, one of us accidentally tried a *telnet* to the outside from his workstation. It shouldn't have worked, but it did. While the machine did have an Ethernet port connected to the gateway LAN, for monitoring purposes (see Section 7.1.1), the transmit leads were cut. How did the packets reach their destination?

It took a lot of investigating before we figured out the answer. We even wondered if there was some sort of inductive coupling across the severed wire ends. But moving them around didn't make the problem go away.

Eventually, we realized the sobering truth: another router had been connected to the gateway LAN, in support of various experiments. It was improperly configured, and emitted a "default" route entry to the inside. This route propagated throughout our internal networks, providing the monitoring station with a path to the outside.

And the return path? Well, the monitor was, as usual, listening in promiscuous mode to all network traffic. When the acknowledgment packets arrived to be logged, they were processed as well.

The incident could have been avoided if the internal network was monitored for spurious default routes, or if our monitoring machine did not have an IP address that was advertised to the outside world.

3.3.10 Routing Filters

By this point, the virtues and limitations of packet filtering should be clear. What is less obvious is that routing information should be filtered as well. The reason is simple: if a node is completely unreachable, it may as well be disconnected from the net. Its safety is almost that good. (But not quite—if an intermediate host that can reach it is also reachable from the Internet and is compromised, the allegedly unreachable host can be hit next.) To that end, routers need to be able to control what routes they advertise over various interfaces.

Consider again the topology shown in Figure 3.9. Assume this time that hosts on NET 2 and NET 3 are not allowed to speak directly to the outside. They are connected to the router so that they can talk to each other, and to the gateway host on NET 1. In that case, the router should not advertise paths to NET 2 or NET 3 on its link to the outside world. Nor should it readvertise any routes that it learned of by listening on the internal links. The router's configuration mechanisms must be sophisticated enough to support this. (Given the principles we have presented here, how should the outbound route filter be configured? Answer: Advertise NET 1 only, and ignore the problem of figuring out everything that should not leak.)

action	src	port	dest	port	flags	comment
allow	SECONDARY	*	OUR-DNS	53		allow our secondary nameserver access
block	*	*	*	53		no other DNS zone transfers
allow	*	*	*	53	UDP	permit UDP DNS queries
allow	NTP.OUTSIDE	123	NTP.INSIDE	123	UDP	ntp time access
block	*	*	*	69	UDP	no access to our tftpd
block	*	*	*	87		the link service is often misused
block	*	*	*	111		No TCP RPC and ...
block	*	*	*	111	UDP	no UDP RPC and no...
block	*	*	*	2049	UDP	NFS. This is hardly a guarantee
block	*	*	*	2049		TCP NFS is coming: exclude it
block	*	*	*	512		no incoming "r" commands ...
block	*	*	*	513		...
block	*	*	*	514		...
block	*	*	*	515		no external lpr
block	*	*	*	540		uucpd
block	*	*	*	6000-6100		no incoming X
allow	*	*	ADMINNET	444		encrypted access to transcript mgr
block	*	*	ADMINNET	*		nothing else
block	PCLAB-NET	*	*	*		anon. students in pclab can't go outside
block	PCLAB-NET	*	*	*	UDP	... not even with FSP and the like!
allow	*	*	*	*		all other TCP is OK
block	*	*	*	*	UDP	suppress other UDP for now

Figure 3.12: Some filter rules for a university. Rules without explicit protocol flags refer to TCP. The last rule, blocking all other UDP service, is debatable for a university.

There is one situation in which "unreachable" hosts can be reached: if the client employs IP source routing. Some routers allow you to disable that feature: if possible, do it. The reason is not just to prevent some hosts from being contacted. An attacker can use source routing to do address-spoofing [Bellovin, 1989]. Caution is indicated: there are bugs in the way some routers and systems block source routing. For that matter, there are bugs in the way many hosts handle source routing; an attacker is as likely to crash your machine as to penetrate it.

Filters must also be applied to routes learned from the outside. This is to guard against *subversion by route confusion*. That is, suppose that an attacker knows that HOST A on NET 1 trusts HOST Z on NET 100. If a fraudulent route to NET 100 is injected into the network, with a better metric than the legitimate route, HOST A can be tricked into believing that the path to HOST Z passes through the attacker's machine. This allows for easy impersonation of the real HOST Z by the attacker.

To some extent, packet filters obviate the need for route filters. If *rlogin* requests are not permitted through the firewall, it does not matter if the route to HOST Z is false—the fraudulent *rlogin* request will not be permitted to pass. But injection of false routes can still be used to subvert legitimate communication between the gateway machine and internal hosts.

action	src	port	dest	port	flags	comment
allow	*	*	MAILGATE	25		inbound mail access
allow	*	*	MAILGATE	53	UDP	access to our DNS
allow	SECONDARY	*	MAILGATE	53		secondary nameserver access
allow	*	*	MAILGATE	23		incoming telnet access
allow	NTP.OUTSIDE	123	NTP.INSIDE	123	UDP	external time source
allow	INSIDE-NET	*	*	*		outgoing TCP packets are OK
allow	*	*	INSIDE-NET	*	ACK	return ACK packets are OK
block	*	*	*	*		nothing else is OK
block	*	*	*	*	UDP	block other UDP, too

Figure 3.13: Some filter rules for a small company. Rules without explicit protocol flags refer to TCP.

As with any sort of address-based filtering, route filtering becomes difficult or impossible in the presence of complex topologies. For example, a company with several locations could not use a commercial data network as a backup to a leased-line network if route filtering were in place; the legitimate backup routes would be rejected as bogus. To be sure, although one could argue that public networks should not be used for sensitive traffic, few companies build their own phone networks. But the risks here may be too great. An encrypted tunnel may be a better solution.

Some people take route filtering a step further: they deliberately use unofficial IP addresses inside their firewalls, preferably addresses belonging to someone else [Rekhter *et al.*, 1994]. That way, packets aimed at them will go elsewhere.

As attractive as this scheme sounds, we don't recommend it. For one thing, it's not neighborly. If you make a mistake in setting up your filter, you risk polluting the global routing tables for the Internet (though it would perhaps draw your attention to your mistake). Another reason is that you risk address collisions as new organizations connect their networks to yours, perhaps by merger or acquisition. (We did *not* have such problems when AT&T acquired NCR, because both companies did use officially-assigned IP addresses.)

Finally, you may convert some day to a different style of firewall, one that lets more packets flow through. If you ever do that, you'll have a massive address conversion problem on your hands, which you're better off avoiding entirely.

3.3.11 Sample Configurations

We cannot give you the exact packet filter for your site, because we don't know what your policies are. But we can give some reasonable samples that may serve as a starting point. The samples in Figures 3.12 and 3.13 are derived in part from CERT recommendations and from our port number table in Appendix B.

A university tends to have an open policy about Internet connections. Still, they should block some common services, such as NFS and TFTP. There is no need to export these services to the world. Also, perhaps there's a PC lab in a dorm that has been the source of some trouble,

so they don't let them access the Internet. (They have to go through one of the main systems that require an account. This gives some accountability.) Finally, there is to be no access to the administrative computers except for access to a transcript manager. That service, on port 444, uses strong authentication and encryption.

On the other hand, a small company with an Internet connection might wish to shut out most incoming Internet access, while preserving most outgoing connectivity. A gateway machine receives incoming mail and provides name service for the company's machines. Figure 3.13 shows a sample filter set.

Remember, we consider packet filters inadequate, especially when filtering at the port level. In the university case especially, they only slow down an external hacker.

3.3.12 Packet-Filtering Performance

You do pay a performance penalty for packet filtering. Routers are generally optimized to shuffle packets quickly. The packet filters take time and can defeat the optimization efforts. But packet filters are usually installed at the edge of an administrative domain. The router is connected by (at best) a *DS1* (T1) line (1.544 Mb/sec) to the Internet. Usually this serial link is the bottleneck: the CPU in the router has plenty of time to check a few tables. This may become a bigger problem as faster communications arrive.

Although the biggest performance hit may come from doing any filtering at all, the total degradation depends on the number of rules applied at any point. It is better to have one rule specifying a network than to have several rules enumerating different hosts on that network. Choosing this optimization requires that they all accept the same restrictions; whether or not that is feasible depends on the configuration of the various gateway hosts. Or you may be able to speed things up by ordering the rules so that the most common types of traffic are processed first. (But be careful; correctness is much more important than speed.) As always, there are trade-offs.

There may also be performance problems if you use a two-router configuration. In such cases, the inside router may be passing traffic between several internal networks as well. Degradation here is not acceptable.

3.3.13 Implementing Packet Filters

There are a number of ways to implement packet filters. The easiest, of course, is to buy a router that supports them. But there are some host-based alternatives.

Digital Equipment Corporation has developed *screend*, a kernel modification that permits a user process to pass on each packet before it is forwarded [Mogul, 1989, 1991]. It is available on a number of operating systems, including, of course, Ultrix. A version for BSDI is promised soon. The code for *screend* is freely available, but with some strings attached that not everyone will find acceptable. Unfortunately, you need source to other parts of the kernel in order to install it.

Some other vendors provide similar functionality. SGI systems have *ipfilterd*, for example.

Of course, there's no need to use a UNIX system as a packet filter. A number of PC-based packages exist, such as TAMU [Safford *et al.*, 1993b] and Karlbridge. As we have noted, you may not need much speed; a surplus unit may work quite well.

3.3.14 Summary

Many advanced gateway designs rely in part on packet filtering. They are likely to work well, but pure packet filters leave us feeling uncomfortable. Some of these designs become ineffective if a vendor software problem compromises the packet filter. We have heard of at least two such software problems to date,[1] although the vendors are very careful about such things. Worse yet, it takes either a conservative stance or a great deal of knowledge about the Internet to design an effective packet filter. Also, there are highly desirable services that cannot be implemented in a pure packet-filtering environment.

We are inclined to place our trust in a simpler design. Packet filters are a useful tool, but they do not leave us with confidence in their correctness and hence their safety.

3.4 Application-Level Gateways

An application-level gateway represents the opposite extreme in firewall design. Rather than using a general-purpose mechanism to allow many different kinds of traffic to flow, special-purpose code can be used for each desired application. Although this seems wasteful, it is likely to be far more secure than any of the alternatives. One need not worry about interactions among different sets of filter rules, nor about holes in thousands of hosts offering nominally secure services to the outside. Only a chosen few programs need be scrutinized.

Application gateways have another advantage that in some environments is quite critical: it is easy to log and control *all* incoming and outgoing traffic. The SEAL package [Ranum, 1992] from Digital Equipment Corporation takes advantage of this. Outbound FTP traffic is restricted to authorized individuals, and the effective bandwidth is limited. The intent is to prevent theft of valuable company programs and data. While of limited utility against insiders, who could easily dump the desired files to tapes or floppies, it is a powerful weapon against electronic intruders who lack physical access.

Electronic mail is often passed through an application-level gateway, regardless of what technology is chosen for the rest of the firewall. Indeed, mail gateways are valuable for their other properties, even without a firewall. Users can keep the same address, regardless of which machine they are using at the time. This book, for example, was composed on no fewer than six different computers, but mail sent from any of them would bear a return address of RESEARCH.ATT.COM. The gateway machines also worry about mail header formats and logging (mail logging is a postmaster's friend) and provide a centralized point for monitoring the behavior of the electronic mail system.

It is equally valuable to route incoming mail through a gateway. One person can be aware of all internal connectivity problems, rather than leaving it to hundreds of random system administrators around the Internet. Reasonably constant mail addresses—*Firstname.Lastname*@ORG.DOMAIN is popular—can be accepted and processed. Different technologies, such as *uucp*, can be used to deliver mail internally. Indeed, the need for incoming mail gateways is so obvious that the DNS

[1] See CERT Advisory CA-92:20, December 10, 1992, and CERT Advisory CA-93:07, April 22, 1993.

has a special feature—`MX` records—defined to support them. No other application has a defined mechanism for indirect access.

These features are even more valuable from a security perspective. Internal machine names can be stripped off, hiding possibly valuable data (see Section 2.3). Traffic analysis and even content analysis and recording can be performed to look for information leaks. But these abilities should be used with the utmost reluctance, for both legal (Chapter 12) and ethical reasons.

Application gateways are often used in conjunction with the other gateway designs, packet filters and circuit-level relays. As we show later (Section 4.5.7), an application gateway can be used to pass X11 through a firewall with reasonable security. The semantic knowledge inherent in the design of an application gateway can be used in more sophisticated fashions. As described earlier, *gopher* servers can specify that a file is in the format used by the *uuencode* program. But that format includes a file name and mode. A clever gateway could examine or even rewrite this line, thus blocking attempts to force the installation of bogus `.rhosts` files or shells with the `setuid` bit turned on.

The type of filtering used depends on local needs and customs. A location with many PC users might wish to scan incoming files for viruses.

We note that the mechanisms just described are intended to guard against attack from the outside. A clever insider who wanted to retrieve such files certainly would not be stopped by them. But it is not a firewall's job to worry about that class of problem.

The principal disadvantage of application-level gateways is the need for a specialized user program or variant user interface for most services provided. In practice, this means that only the most important services will be supported. This may not be entirely bad—again, programs that you do not run cannot hurt you—but it does make it harder to adopt newer technologies. Also, use of such gateways is easiest with applications that make provision for redirection, such as mail and X11. Otherwise, new client programs must be provided.

3.5 Circuit-Level Gateways

The third type of gateway—our preference for outgoing connections—is circuit level. Circuit gateways relay TCP connections. The caller connects to a TCP port on the gateway, which connects to some destination on the other side of the gateway. During the call the gateway's relay program(s) copy the bytes back and forth: the gateway acts as a wire.

In some cases a circuit connection is made automatically. For example, we have a host outside our gateway that needs to use an internal printer. We've told that host to connect to the print service on the gateway. Our gateway is configured to relay that particular connection to the printer port on an internal machine. We use an access control mechanism to ensure that only that one external host can connect to the gateway's printer service. We are also confident that this particular connection will not provide a useful entry hole should the external host be compromised.

In other cases, the connection service needs to be told the desired destination. In this case, there is a little protocol between the caller and the gateway. This protocol describes the desired destination and service, and the gateway returns error information if appropriate. In our implementation, called *proxy*, the destination is a host name. In *socks* (discussed later), it is the numeric

IP address. If the connection is successful, the protocol ends and the real bytes start flowing. These services require modifications to the calling program or its library.

In general, these relay services do not examine the bytes as they flow through. Our services do log the number of bytes and the TCP destination. These logs can be useful. For example, we recently heard of a popular external site that had been penetrated. The Bad Guys had been collecting passwords for over a month. If any of our users used these systems, we could warn them. A quick *grep* through the logs spotted a single unfortunate (and grateful) user. Chapter 11 shows some statistical information we have gathered from our proxy logs.

The outgoing proxy TCP service provides most of the Internet connectivity our internal users need. As noted, though, protocols such as FTP and X11 require incoming calls. But it is too much of a security risk to permit the gateway to make an uncontrolled call to the inside.

Any general solution is going to involve the gateway machine listening on some port. Though we defer discussion of the details until the following chapter, this approach demonstrates a subtle problem with the notion of a circuit gateway: uncooperative inside users can easily subvert the intent of the gateway designer, by advertising unauthorized services. It is unlikely that, say, port 25 could be used that way, as the gateway machine is probably using it for its own incoming mail processing, but there are other dangers. What about an unprotected *telnet* service on a nonstandard port? An NFS server? A multiplayer game? Logging can catch some of these abuses, but probably not all.

Clearly, some sorts of controls are necessary. These can take various forms, including a time limit on how long such ports will last (and a delay before they may be reused), a requirement for a list of permissible outside callers to the port, and even user authentication on the setup request from the inside client. Obviously, the exact criteria depend on your stance.

The other big problem with circuit relays is the need to provide new client programs. Although the code changes are generally not onerous, they are a nuisance. Issues include availability of application source code for various platforms, version control, distribution, and the headache to users of having to know about two subtly different programs.

Several strategies are available for making the necessary changes. The best known is the *socks* package [Koblas and Koblas, 1992]. It consists of a set of almost-compatible replacements for various system calls: `socket`, `connect`, `bind`, etc. Converting an application is as simple as replacing the vanilla calls with the *socks* equivalents. A version of it has been implemented via a replacement shared library, similar to that used in *securelib* [LeFebvre, 1992] and *3-D FS* [Korn and Krell, 1989]. This would permit existing applications to run unchanged. But such libraries are not portable, and it may not be possible to include certain of the security features mentioned earlier.

Our own approach is somewhat different. We made a simple change to the IPC library described in Section 6.1. Instead of writing

```
fd = ipcopen("tcp!desired.host!portnum", "");
```

a programmer can now write

```
fd = ipcopen("proxy!desired.host!portnum", "");
```

to make a call through the gateway. Naturally, this call recurses through the normal path selection mechanism; thus, the path to the gateway could use the Datakit VCS (`dk`) instead of TCP:

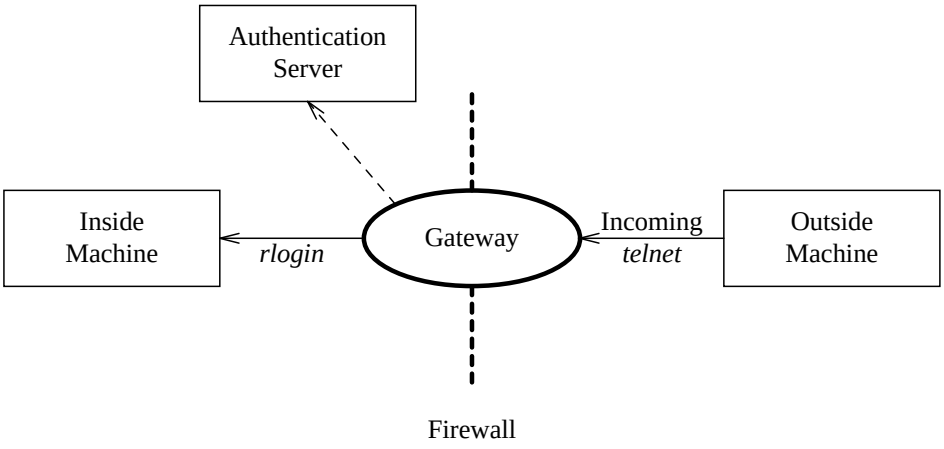


Figure 3.14: Call flow diagram for incoming *telnet* access.

```
fd = ipcopen("dk!nj/astro/gateway.relay", "");
```

Application and circuit gateways are well suited for some UDP applications. The client programs must be modified to create a virtual circuit to some sort of proxy process; the existence of the circuit provides sufficient context to allow secure passage through the filters. The actual destination and source addresses are sent in-line. However, services that require specific local port numbers are still problematic.

3.6 Supporting Inbound Services

Regardless of the firewall design, it is generally necessary to support various incoming services. These include things like electronic mail, FTP, logins, and possibly site-specific services. Naturally, access to any of these must be blessed by the filter and the gateway.

The most straightforward way to do this is to provide these services on the gateway itself. This is the obvious solution for mail and FTP. For incoming logins, we provide a security server; users must have one-time password devices to gain access to inside machines. If they pass that test, the gateway program will connect them to an inside machine, using some sort of preauthenticated connection mechanism such as *rlogin* (Figure 3.14).

Ganesan has implemented a gateway that uses Kerberos to authenticate calls [Ganesan, 1994]. Once the gateway has satisfied itself about the identity of the caller, it will pass the connection on to the desired internal server. This scheme assumes that the external Kerberos server is secure, because anyone who penetrates it will be able to spoof the firewall.

Regardless of the scheme used, all incoming calls carry some risk. The *telnet* call that was authenticated via a strong mechanism could be the product of a booby-trapped command. Consider, for example, a version that, after a few hundred bytes, displays "`Destination`

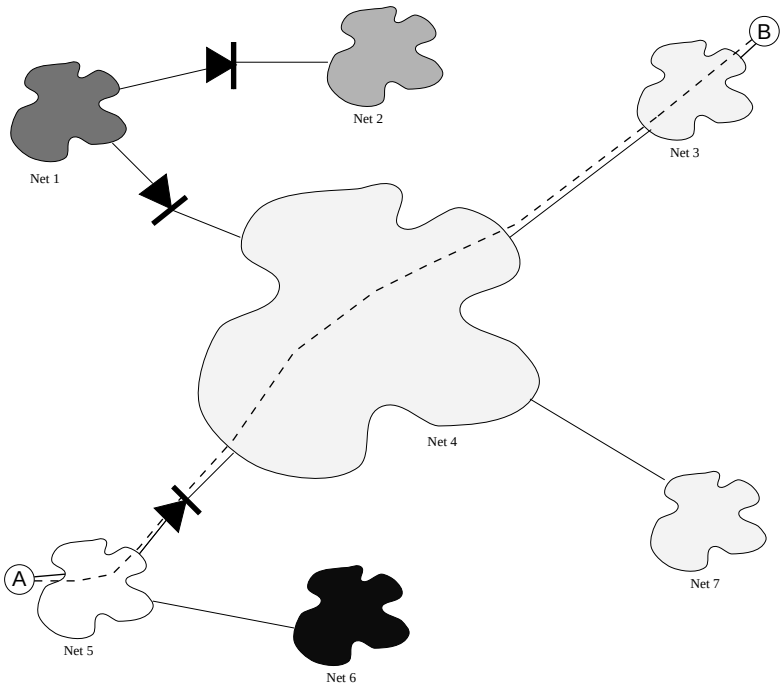


Figure 3.15: Tunneling past a firewall.

`Unreachable`" on the console and exits—but before doing that, forks, and retains the open session to your inside machine. Similarly, a legitimate user who connects for the purpose of reading mail takes the risk that some of those messages contain sensitive information, information that can now be read by anyone monitoring the unprotected, untrustworthy outside network.

3.7 Tunnels Good and Bad

Although firewalls offer strong protection, *tunnels* (Figure 3.15) can be used to bypass them. As with most technologies, tunnels can be used in good or bad ways.

Tunneling refers to the practice of *encapsulating* a message from one protocol in another, and using the facilities of the second protocol to traverse some number of network hops. At the destination point, the encapsulation is stripped off, and the original message is reinjected into the network. In a sense, the packet burrows under the intervening network nodes, and never actually sees them. There are many uses for such a facility, such as encrypting links and supporting mobile hosts. More are described in [Bellovin, 1990].

In some cases, a protocol may be encapsulated within itself. That is, IP may be buried within either IP or some part of its own protocol suite, such as TCP or UDP. That is the situation we are concerned about here. If a firewall permits user packets to be sent, a tunnel can be used to bypass the firewall. The implications of this are profound.

Suppose that an internal user with a friend on the outside dislikes the firewall, and wishes to bypass it. The two of them can construct (dig?) a tunnel between an inside host and an outside host, thereby allowing the free flow of packets. This is far worse than a simple outgoing call, since incoming calls are permitted as well.

Almost any sort of mechanism can be used to build a tunnel. At least one vendor of a *Point-to-Point Protocol* (*PPP*) package [Simpson, 1992] supports TCP tunneling. There are reports of *telnet* connections and even DNS messages being used to carry IP packets. Almost any gateway that supports anything more powerful than mail relays can be abused in this fashion (but see RFC 1149 [Waitzman, 1990]). Even pairs of FTP file transfer connections can provide a bidirectional data path.

The extent of the damage done by a tunnel depends on how routing information is propagated. As noted earlier, denial of routing information is almost as effective as full isolation. If the tunnel does not leak your routes to the outside, the damage is less than might be feared at first glance. On the other hand, routing filters are difficult to deploy in complex topologies; if the moles choose to pass connectivity information, it is hard to block them. In the Internet, the backbone routers do, in fact, perform filtering. Thus, if your internal networks are not administratively authorized for connection to the Internet, routes to them will not propagate past that point. Even so, you are exposed to anyone using the same network provider as the tunnel exit.

Often, such a situation can be detected. If you are using an application- or a circuit-level gateway, and an external router knows a path to any internal network except the gateway's, *something* is leaking. This argues strongly that a gateway net should *not* be a subnet of an internal net. Rather, it should have its own, separate, Class C address. Standard network management tools may be able to hunt down the source, at which time standard people management tools should be able to deal with the root cause. Unauthorized tunnels are, in the final analysis, a management problem, not a technical one. If insiders will not accept the need for information security, firewalls and gateways are likely to be futile. (Of course, it is worth asking if your protective measures are too stringent. Certainly, that happens as well.) Once suspected or spotted, the gateway logging tools should be able to pick out the tunnels.

Tunnels have their good side as well. When properly employed, they can be used to bypass the limitations of a topology. For example, a tunnel could link two separate sites that are connected only via a commercial network provider. Firewalls at each location would provide protection from the outside, while the tunnel provides connectivity. If the tunnel traffic is encrypted (see Section 13.4), the risks are low and the benefits high.

3.8 Joint Ventures

A principal disadvantage of firewalls is that they are "all or nothing" devices. Often, though, the real world is more complex. Companies often wish to let support personnel from vendors connect

in order to diagnose problems. Or they may be engaged in limited joint venture agreements with other companies. The two situations are quite similar, though less special-purpose setup can be done for the former.

In a typical joint venture agreement, two or more companies agree to work together on some specific project. Often, they are competitors in other fields. Naturally, they will require access to shared computer resources. The problem is how to set this up in a secure fashion, with respect to several criteria.

First, of course, the shared machines still require protection from outsiders. Indeed, that need may be greater; often, the very existence of a partnership may be confidential. Thus, some sort of firewall is mandatory.

Second, one must assume that the two partners do not fully trust each other. It thus may be necessary to isolate the shared machines from the internal networks of the partners.

Third—and this is the catch—it is desirable that users have high-quality access to both the shared machines and to their own company's home machines.

Finally, of course, there is the question of whether or not the other party's machines have been compromised. For this reason, we focus on solutions that are workable even if both parties use firewalls.

Much rides on the precise definition of "high quality." Often, it means editing with one's standard editing tools, compiling the same way, etc. To the extent that will suffice, the problem can be solved by using some sort of shared file system.

The ideal shared file system would be a connection-based TCP-level technology. The (possibly encrypted) TCP circuit could be tunneled out one gateway and into another domain, where a user-level server could operate in a `chroot` environment in the shared file system. We know of three such network file systems: Peter Weinberger's research UNIX Netb file system [Rago, 1990], the *Remote File System* (*RFS*), and NFS Version 3. None is generally available. NFS Version 2 is the only answer.

Since plain NFS is not sufficiently secure, and would not easily pass through a firewall in any event, we have developed a proxy NFS scheme using TCP. Because it uses TCP, it is compatible with our other proxy services. Because we wrote our own initialization functions, we were able to provide strong authentication and optional encryption. Finally, we can both import and export file systems, with a high degree of safety. Details are given in Chapter 4.

The *Truffles* project [Cook *et al.*, 1993; Reiher *et al.*, 1993; Cook and Crocker, 1993a, 1993b] is another possible solution. It extends NFS to permit shared network directories. PEM-based cryptography (see Section 13.5.3) is used for authentication and confidentiality. Both sides may have copies of shared files; if an update conflict occurs, it is resolved manually.

Another solution to the joint venture problem uses an isolated subnet within one company's network. A firewall prevents any *outgoing* calls from it. Both companies use tunnels to connect to machines on that network. Since these are incoming calls, they can pass through the firewall. But no outgoing calls can, thus protecting the rest of the host company's machines.

A third solution is not practical today in most environments, but may be in the future. It relies on the use of so-called *multilevel secure* hosts and routers [Amoroso, 1994], i.e., machines that support security labels for processes and network packets. The outside users tunnel through the firewall to the shared machine. On that machine, their processes are given a special label, one that

lacks a category present on all nonshared files on the machine and on the network interface. As a result, they can neither read any of those files, nor establish any network connections. If the joint venture involves more than one machine, then the network connecting the entire set of machines can be labeled to permit communication; however, the router port feeding that network would not be. Their messages would thus be confined to the designated area.

This solution, unlike the previous two, is suitable for casual visitors as well. Unfortunately, very few networked multilevel secure systems are available today.

3.9 What Firewalls Can't Do

> "Thought-screens interfered so seriously with my methods of procedure," the Palainian explained, "that I was forced to develop a means of puncturing them without upsetting their generators. The device is not generally known, you understand."
>
> *Nadreck of Palain VII* in *Second Stage Lensman*
> —E.E. "DOC" SMITH

Firewalls are a powerful tool for network security. However, there are things they cannot do. It is important to understand their limitations as well as their benefits.

Consider the usual network protocol layer cake. By its nature, a firewall is a very strong defense against attacks at a lower level of the protocol stack. For example, hosts behind a circuit-level relay are more or less immune to network-level attacks, such as IP address-spoofing. The forged packets cannot reach them; the gateway will only pass particular TCP connections that have been properly set up.

In contrast, firewalls provide almost no protection against problems with higher level protocols, except by peeking. The best TCP relay in the world is no protection if the code that uses it is buggy and insecure. You only get protection at this level if your gateway refuses to connect you to certain services (i.e., X11), and even that decision is applying application-layer knowledge to make that decision. (If you think of the standard protocol stack as an onion rather than as a layer cake, peering up through the layers may be referred to as "looking through a glass onion.")

The most interesting question is what degree of protection a firewall can provide against threats at its own level. The answer turns entirely on how carefully the gateway code—the permissive part—is written. Thus, a mail gateway, which runs at the application level, must be exceedingly careful to implement all of the mail protocols, and all of the other mail delivery functions, absolutely correctly. To the extent that it is insecurely written—*sendmail* comes to mind—it cannot serve as an adequate firewall component.

The problems, however, do not stop there. *Any* information that passes inside can trigger problems, if a sensitive component should lay hands (or silicon) on it. We have seen files that, when transferred over a communications link, effectively brought down that link, because of bit pattern sensitivity in some network elements. Were that deliberate, we would label it a denial-of-service attack.

A recent *sendmail* bug[2] provides a sterling example. Problems with certain mail header lines could tickle bugs in delivery agents. Our firewall, and many others, paid almost no attention to headers, believing that they were strictly a matter for mail readers and composers (known as *user agents* in the e-mail biz). But that meant that the firewalls provided no protection against this problem, because under certain circumstances, *sendmail*—which is run on many internal machines here—does look at the headers, and certain entries made it do evil things.

Furthermore, even if we had implemented defenses against the known flaws, we would still be vulnerable to next year's. If someone invented a new header line that was implemented poorly—and this particular problem did involve a nonstandard header—we would still be vulnerable. We could have protected ourselves if and only if we had refused to pass anything but the minimal subset of headers we did know of, and even then there might have been danger if some aspect of processing a legitimate, syntactically correct header was implemented poorly. At best, a firewall provides a convenient single place to apply a corrective filter.

[2]See CERT Advisories CA-93:15, October 21, 1993; CA-93:16, November 4, 1993; and CA-93:16a, January 7, 1994.