11

Firewall Engineering

Once upon a time, all firewalls were hand-constructed, perhaps from software obtained from various pioneers at DEC and TIS. For these early gateways, packet filtering was easy, but not very sophisticated, which meant that it was not very safe. There were no tools to keep track of TCP sessions at the packet level. (Two of us, Steve and Bill, designed a dynamic packet filter in September, 1992, based mostly on off-the-shelf components, but the implementation looked complex enough that it scared us off.)

Gateways back then were mostly at the application level. We built filters for FTP and SMTP access. Circuit gateways allowed modified clients to make connections to the Internet without IP connectivity—between intranet and Internet were computers and programs that simulated wires. This lack of direct IP connectivity bought a great deal of security. Tricks with IP fragmentation and other firewalking operations were not possible, and corporate gateways in particular could be quite high-grade. Admittedly, such tricks hadn't been invented, but that's not the point—we were trying to protect ourselves against unknown attacks.

This early approach (taken partly at our urging in the first edition of this book) has left a legacy in many large corporate intranets. The lack of IP connectivity created a culture of separation, IP addresses were assigned with abandon, and there was often a (false) sense of safety behind highly restrictive firewalls.

Today's intranets are too large to rely mainly on perimeter defenses. You simply don't know the extent of your network if it is larger than a few dozen hosts.

Most people don't build their own firewalls these days; they buy them, and (generally) rightly so. We have encountered astonishment from network administrators at the suggestion that they might build their own, as if we were suggesting that they design a do-it-yourself fuel-injection system for their own car.

In fact, it can be easy to construct a strong firewall. A number of open-source operating systems are very reasonable, trusted computing bases, and most of the typical firewall functions are available in their kernels. A variety of proxies are easily obtained and run efficiently in user mode. Modern hardware can easily keep up with heavy traffic flows.

Parts of a modern firewall may be implemented like our old application-level gateways, but usually they operate at level three, the IP-packet level. Some work as filtering bridges at level two, examining the contents of Ethernet packets. These devices may offer the ultimate in transparency, as they could have no IP address associated with them at all. Bridge-level firewalls may be dropped into a connection without reconfiguring a router.

We don't describe this process in detail here, but we do discuss the basic design and engineering decisions involved. These concepts are useful in evaluating commercial firewalls, as well as constructing simple, efficient ones.

It is not clear which choice offers more security. It is possible to build a highly attack-resistant, efficient firewall quite easily. It is harder to add the variety of application-level filters that commercial systems offer. Web and mail proxies add complexity, and filters to detect specific viruses require teams of experts to keep the virus descriptions and engines up-to-date. Furthermore, many of the commercial proxies are for protocols for which open documentation is unavailable. (Of course, as we've noted, that begs the question of whether or not you need to pass a given protocol through your firewall, and hence whether or not you even need a proxy.) The documentation of a commercial firewall may be better, and one can get help from user groups and Web pages.

We are going to implement simple policies, which may apply to a variety of configurations—from corporate gateways to a firewall in the home. The principle of least privilege and *keep it simple, stupid (KISS)* are just as important in firewall configuration as in other security pursuits. In addition, only permit the minimum number of services through, and try to understand them well. Only trust a minimum number of auxiliary hosts.

This is often not the practice today. We know of companies that have installed rulesets in a single firewall with *thousands* of rules and thousands of host groups. It can take days for an analyst to try to understand the underlying policies—and we emphasize *try*. Tools such as Fang [Mayer *et al.*, 2000] can help, but this level of complexity is way out of control, and the "firewall" might be better implemented as a wire that lets all the traffic through. Certainly the administration costs would be much lower.

11.1 Rulesets

Firewalls and similar devices are configured with rulesets. These may be entered with a graphical user interface (see the Sidebar on page 213), or simply entered as a series of text commands. Once you have seen one of these command sets, the others should be relatively easy to figure out. The syntax varies a bit, but surprisingly, so does the processing.

These rulesets generally consist of a verb and a pattern. A very simple set might be

```
permit incoming smtp
permit outgoing TCP
log incoming netbios
block all
```

An incoming packet is tested according to the rules. We run through the rules starting at the top, and when we learn how to dispose of the packet, we stop. Here we have three verbs, `permit`, `block`, and `log`. `Permit` and `block` tell us how to dispose of the packet, so we can stop

Graphical User Interfaces

Since the mid-1990s, it has become *de rigueur* to have a *graphical user interface (GUI)* to configure firewalls and similar network elements. The developers say that the marketers require it. The marketers say that the customers demand it, because it makes firewall configuration easier. We think the customers are mistaken, and here's why.

GUIs, with drop-down menus, are the most common interface available on computers. The X Window System and Apple's early Macintosh designs work well for many applications, such as moving files around in folders. GUIs work best for data that is amenable to graphical display. There are many visualizations for which GUIs are easily the best option known.

We have never seen a good graphical visualization for firewall rules and policies. True, you could show hosts and privilege groups in a graphic display, and use links to display relations between them. But these privilege relationships can involve complicated specifications: There are too many ports, too many protocols, and too many conditions we might wish to encode into our policies. Without the visualization, the "graphical user interface" becomes a forms entry program. Although a form is not a bad way to enter a stereotypical bank record, it doesn't let us express relationships well.

What is the alternative? A configuration file written in a high-level language answers these needs nicely. The firewall policy and conditions are expressed as a series of commands, conditionals, and definitions written in a simple language. If you are unfamiliar with the language, the vendor can supply sample files containing comment lines with explanations. These sample files can contain typical configurations for various situations that might apply.

If the language is decent, the rules are easy to read. The file can be scanned with a familiar text editor. The user already knows how to move rules around and make global changes. The editor can be as simple as Wordpad or the text entry window on a browser. True, the configuration file has to be scanned for errors at some point, whereas a GUI can (usually) catch the errors as they are typed.

A GUI has to provide special summary screens to show relevant information for each record, plus special screens to show the details of each record, because it doesn't all fit on a single screen line. GUIs tend to add a great deal of development time, and should require visualization experts to help make the interface understandable and useful.

Finally, people argue that GUIs make the firewall administrator's job easier. Although we disagree—we've found that GUIs get in the way of configuring a firewall quickly—we don't think the hard part of firewall administration is data entry, it is knowing what the appropriate policies are. That a GUI would make an important job simpler is a dangerous claim. You need to know what you are doing for almost anything but a trivial firewall policy. At best, GUIs are novice-friendly, but expert-hostile.

processing right there. An incoming SMTP packet would be accepted according to the first rule, and no further tests are necessary. The `log` verb tells the firewall to record information about the packet, but doesn't tell us whether to accept or deny it, so the processing continues. Hence, our log would contain only incoming attempts to connect to *netbios*. Packets of all sorts other than incoming SMTP and outgoing TCP are blocked.

This general approach to processing a packet seems obvious to us, so it is surprising that some filters do not implement this simple top-down approach. Some have tried rearranging the rules to speed up packet filter processing. Others process the packet through all the rules and then decide. This is confusing, and it is very important not to confuse the network administrator. Configuration errors are the chief source of firewall failures. (We distinguish this sort of failure from failures of the *policy*, where the administrator mistakenly *decides* to let some packets pass, without realizing the danger.)

In general, these languages describe individual packets, but they can describe connections, and even entire service suites. The endpoints may be hosts, networks, or interfaces on the firewall or router. This description problem is similar for firewalls, intrusion detection systems, sniffers, and anything else that is trying to deal with Internet traffic above the simple routing level. They could be quite fancy and powerful. If you implement such a language, make sure that the casual network administrator can understand and use it, as he or she may not be conversant with object-oriented modules and the like.

The *Berkeley packet filter (BPF)* has a packet selection language. So does *tcpdump*. Cisco routers implement one for packet filtering, as do *ipf*, *ipfw*, *Network Flight Recorder (NFR)*, *bro*, and *ipchains* (under Linux.) Most of these apply the rules in the order given, but not all.

A packet can be filtered in three places as it transits through the filtering device: at the incoming interface, during the routing computation, and on the way out on the outgoing interface. In most firewall configurations, the network an interface connects to has a particular security level and function. A typical corporate firewall might have a total of three interfaces, one each for the Internet, the intranet, and the DMZ. Much of a packet's processing depends on its provenance. We want to check packets from the Internet for all kinds of nastiness: spoofing of inside or local addresses, weird fragmentation, and so on. A DMZ's interface should be much simpler. Only a few packet flows are expected, and they should be well-mannered. We should log any unusual activity.

One would hope that packets from the internal network would be well-behaved, but they probably aren't. Aside from a sea of misconfiguration and routing problems, internal hosts might be infected with worms or viruses, or operated by adversarial users. It is also good practice to limit the damage that an internal attacker can do to the firewall itself—a firewall should be no more susceptible to an attack from a high-security network than from a low-security network.

11.2 Proxies

Packet filters either accept packets, block them, or forward the packets to a different port (possibly on another machine) for a proxy to handle. Proxies can be used to make filtering decisions based on information above the packet layer or above the entire transport layer. They are also used to define very simple packet filtering rules, while handing off the complexity to someone else.

Accepting arbitrary UDP packets through a firewall is a bad idea. However, many programs that users demand, such as audio streaming or NetMeeting, communicate over UDP. One way to enable this service but still disallow UDP through the firewall is to proxy the service. Most sites allow outbound TCP connections, so users connect to an external proxy over TCP. The external proxy speaks TCP to the user and UDP to the service. From the server's point of view, it is speaking with a regular UDP client. From the user's point of view, and more importantly, the firewall's, there is a normal TCP connection from the user to the proxy. The job of the proxy is to translate the two connections for each other so that the communication works.

Proxies can be specified within an application, in which case the program must support the use of proxies. Firewalls can also implement transparent proxies that intercept requests from clients based on port number. These automatically forward packets to a proxy program, possibly on a different port on another machine. The client need not be aware of the proxy.

An example of a proxy is DUAL Gatekeeper, which proxies H.323 and allows NetMeeting from behind a firewall. While most H.323 programs use TCP ports 1720 and 1731 for control messages, the media data is sent via RTP [Schulzrinne *et al.*, 1996] over UDP, with dynamic port numbers. Without a proxy, it is impossible to allow H.323 traffic and still maintain a reasonable firewall policy in a stateless packet filter.

11.3 Building a Firewall from Scratch

Though this may sound daunting to the novice, it isn't very hard, and doesn't have to take much time. In a recent emergency, we built and installed a solid, state-of-the-art, NAT-ing firewall in two hours starting with an empty computer and a recent FreeBSD installation CD.

In this section, we look at three different firewalls one might build. The first shows that it is quite simple to configure a personal firewall for Linux. We use the *ipchains* program to set up a firewall with the policy described below. The second example shows how to set up an organization's firewall with a DMZ, using the *ipf* program. Finally, we discuss application-based filtering, which of course only makes sense in the context of a host.

We start with a security policy. This doesn't have to be a thick book of regulations that nobody reads. A series of simple guidelines should do. And remember that reasonable people can disagree on the risks and benefits of particular decisions.

Here is a relatively minimal, and typical, policy. Internal users are trusted, and permitted nearly unhampered access to the Internet. They are explicitly allowed to

- initiate outgoing TCP connections,
- run *ping* and *traceroute*,
- issue DNS queries, and
- set their clock using an external time server.

Insiders may not offer any Internet services to the outside world. This means that on a household network, e-mail is obtained by polling. Incoming services must be implemented by explicit gateways. For example, incoming mail would have to go through a mail server. There is no

UDP service allowed through the gateway with the exception of the explicit packets needed to implement this policy.

The outside world should not be able to initiate any access to the internal network.

(This policy is a fine first-cut at a security policy, but it leaves a lot of possible holes. For example, the TCP policy allows users to connect to external POP3 servers, and perhaps import viruses. Chapter 10 discusses these issues in more detail.)

For the first example, we look at protecting a personal Linux box with simple firewall rules that define this policy.

11.3.1 Building a Simple, Personal Firewall

Ipchains is a Linux program that acts as a general-purpose stateless packet filter. The code is a descendent of *ipfw* in BSD, and is available from `http://netfilter.filewatcher.org/ipchains`, and other places. *Iptables* is another program that is very similar in nature to *ipchains*—the main differences are in the syntax accepted. Both of these programs are very expressive—they can be used to provide NAT service, route packets, and, of course, filter traffic based on port numbers, addresses, and flags. The *iptables* program groups firewall rules into *chains*, which are simply collections of rules that go together logically. There are three system chains: *input*, *forward*, and *output*. Input and output are used to make decisions when packets enter and leave an interface, respectively. The *forward* chain is used for routing decisions, or in *ipchains*-speak, for *masquerading*.

The chains reside in the kernel, and can be created at startup. There is also a useful utility (coincidentally called *ipchains*) for managing them on the command line. With the *ipchains* utility, the rules take effect immediately; no *init* scripts need to run. The rules are evaluated in order, and the first match disposes of the packet.

Besides the system chains, users can define chains. The user-defined chains also represent logical groupings of rules, which can help keep them organized. For example, there might be a set of rules designed to accept ICMP packets. All of these rules can be grouped into a chain called *icmp-accept*. Then, for example, in the input chain, you could place a rule that sends the packet to be processed by the icmp-accept chain whenever an ICMP packet is encountered. This affords the opportunity for modular and readable rulesets without the clutter of all of the individual rules that are needed. In addition, users can easily share by exchanging chains of rules that are specific to a given subpolicy.

For a wonderful guide on getting started with and configuring *ipchains*, see `http://www.tldp.org/HOWTO/IPCHAINS-HOWTO.html`. This section describes how to set up the personal firewall policy described earlier.

Note that this is a firewall designed to protect a single computer; it's not a gateway firewall. Thus, we could ignore binding chains to particular interfaces.

The first thing to do when setting up *ipchains* is to make sure that it is not already installed. It is possible that a machine already has rules set up because of default settings, or perhaps you have inherited a laptop from someone else. Typing

```
ipchains -L
```

will show you if any rules are loaded. If there are, you can type

```
ipchains -F
```

to flush out the rules in all the chains. (Note, of course, that this turns off all your filtering... You may want to disconnect from the Net while doing your editing.) Keep in mind that if rules are already in place, the changes you make will disappear the next time you restart; ultimately, you have to make the changes permanent by editing the appropriate start-up script.

For simplicity, we limit the example to the input chain and do not do any forwarding or output filtering. Of course, without any forwarding (masquerading), it doesn't matter whether you use the input or output chain. In our example, we have a host called RUBINLAP. Its IP address is 135.207.10.208. The first commands are as follows:

```
ipchains -A input -j ACCEPT -p TCP -s 135.207.10.208
ipchains -A input -j ACCEPT -p TCP ! -y -d 135.207.10.208
```

"-A input" adds a rule to the *input* chain, and "-s" and "-d" specify source and destination addresses, respectively. "-y" matches packets with the TCP SYN bit set, and the "!" negates the following parameter. Thus, the first rule allows outbound TCP traffic (including connection initiation), and the second rule allows inbound TCP traffic, except for connection initiation. *Ipchains* is not stateful; otherwise, we could just allow outbound SYN packets, and all traffic on the resulting connection. Note the these rules can subject us to firewalking probes (see Section 11.4.5). *Ipchains* doesn't offer a solution to this.

```
ipchains -A input -j ACCEPT -p UDP -d 135.207.10.208 -s 0/0 domain
ipchains -A input -j ACCEPT -p UDP -s 135.207.10.208 -d 0/0 domain
ipchains -A input -j ACCEPT -p UDP -d 135.207.10.208 -s 0/0 ntp
ipchains -A input -j ACCEPT -p UDP -s 135.207.10.208 -d 0/0 ntp
```

These rules allow for DNS and NTP traffic in both directions. This is the only UDP traffic we allow:

```
ipchains -A input -j ACCEPT -p ICMP -s 135.207.10.208 -d 0/0 --icmp-type ping
ipchains -A input -j ACCEPT -p ICMP -s 135.207.10.208 -d 0/0 --icmp-type pong
ipchains -A input -j ACCEPT -p ICMP -d 135.207.10.208 --icmp-type ping
ipchains -A input -j ACCEPT -p ICMP -d 135.207.10.208 --icmp-type pong
ipchains -A input -j ACCEPT -p ICMP -d 135.207.10.208 --icmp-type time-exceeded
ipchains -A input -j ACCEPT -p ICMP -d 135.207.10.208 --icmp-type
                                                        fragmentation-needed
```

We allow ourselves to ping and be pinged. The name "pong" identifies ICMP Echo Reply packets. We allow inbound ICMP Time exceeded messages so that we can run *traceroute*. The ICMP Fragmentation Needed message is used for MTU discovery, which avoids black holes:

```
ipchains -A input -j ACCEPT -p TCP -y -d 135.207.10.208 auth
```

This rule opens inbound port 113 for the *ident* service: there are abbreviated versions that have no possibility of compromise. Some curious mailers will timeout waiting for a response to an *ident* query; simply returning a TCP RST will help them progress:

```
ipchains -A input -j DENY -l
```

Everything else is denied and logged ("-l"). After these commands are all run, to populate the kernel with filtering rules, the *ipchains -L* command prints out a nice listing of the current rules:

```
Chain input (policy ACCEPT):
target     prot opt     source            destination       ports
ACCEPT     tcp  -y----  rubinlap anywhere                   any ->   any
ACCEPT     tcp  ------  rubinlap anywhere                   any ->   any
ACCEPT     tcp  !y----  anywhere             rubinlap       any ->   any
ACCEPT     udp  ------  anywhere             rubinlap       domain ->   any
ACCEPT     udp  ------  rubinlap anywhere                   any ->   domain
ACCEPT     udp  ------  anywhere             rubinlap       ntp ->   any
ACCEPT     udp  ------  rubinlap anywhere                   any ->   ntp
ACCEPT     icmp ------  rubinlap anywhere                   echo-request
ACCEPT     icmp ------  rubinlap anywhere                   echo-reply
ACCEPT     icmp ------  anywhere             rubinlap       echo-request
ACCEPT     icmp ------  anywhere             rubinlap       echo-reply
ACCEPT     icmp ------  anywhere             rubinlap       time-exceeded
ACCEPT     icmp ------  anywhere             rubinlap       fragmentation-needed
ACCEPT     tcp  -y----  anywhere             rubinlap       any ->   auth
DENY       all  ----l-  anywhere             anywhere          n/a
Chain forward (policy DENY):
Chain output (policy ACCEPT):
```

There are also two useful utilities for saving and restoring rulesets in a chain: *ipchains-save* and *ipchains-restore*. For the preceding ruleset, *ipchains-save input* prints out

```
:input ACCEPT
:forward DENY
:output ACCEPT
Saving 'input'.
-A input -s 135.207.10.208/255.255.255.255 -d 0.0.0.0/0.0.0.0 -p 6 \
      -j ACCEPT -y
-A input -s 135.207.10.208/255.255.255.255 -d 0.0.0.0/0.0.0.0 -p 6 \
      -j ACCEPT
-A input -s 0.0.0.0/0.0.0.0 -d 135.207.10.208/255.255.255.255 -p 6 \
      -j ACCEPT ! -y
-A input -s 0.0.0.0/0.0.0.0 53:53 -d 135.207.10.208/255.255.255.255 -p 17 \
      -j ACCEPT
-A input -s 135.207.10.208/255.255.255.255 -d 0.0.0.0/0.0.0.0 53:53 -p 17 \
      -j ACCEPT
-A input -s 0.0.0.0/0.0.0.0 123:123 -d 135.207.10.208/255.255.255.255 -p 17 \
      -j ACCEPT
-A input -s 135.207.10.208/255.255.255.255 -d 0.0.0.0/0.0.0.0 123:123 -p 17 \
      -j ACCEPT
-A input -s 135.207.10.208/255.255.255.255 8:8 -d 0.0.0.0/0.0.0.0 -p 1 \
      -j ACCEPT
-A input -s 135.207.10.208/255.255.255.255 0:0 -d 0.0.0.0/0.0.0.0 -p 1 \
      -j ACCEPT
-A input -s 0.0.0.0/0.0.0.0 8:8 -d 135.207.10.208/255.255.255.255 -p 1 \
      -j ACCEPT
-A input -s 0.0.0.0/0.0.0.0 0:0 -d 135.207.10.208/255.255.255.255 -p 1 \
```

```
      -j ACCEPT
-A input -s 0.0.0.0/0.0.0.0 11:11 -d 135.207.10.208/255.255.255.255 -p 1 \
      -j ACCEPT
-A INPUT -s 0.0.0.0/0.0.0.0 3:3 -d 135.207.10.208/255.255.255.255 4:4 -p 1 \
      -j ACCEPT
-A input -s 0.0.0.0/0.0.0.0 -d 135.207.10.208/255.255.255.255 113:113 -p 6 \
      -j ACCEPT -y
-A input -s 0.0.0.0/0.0.0.0 -d 0.0.0.0/0.0.0.0 -j DENY -l
```

which can be piped to a file and then restored from later. (These lines were folded to fit on the page.) For some reason, although CIDR format can be used in the *ipchains* command, the save command prints things out using bit masks. Because our example does not use any / addresses, 255.255.255.255 is used. This is no big deal, but it is a bit confusing.

In practice, the last rule will probably log too much information, such as broadcast packets, blasts from runaway processes, and other Internet cruft. One alternative is to add separate rules to log those things that you want to monitor. For example, if you are curious about connection attempts to *irc*, *ssh*, or *telnet*, you could use the following four commands:

```
ipchains -A input -j DENY -p TCP -d 135.207.10.208 irc -l
ipchains -A input -j DENY -p TCP -d 135.207.10.208 ssh  -l
ipchains -A input -j DENY -p TCP -d 135.207.10.208 telnet -l
ipchains -A input -j DENY
```

Attempts to connect to *irc*, *ssh*, and *telnet* on the machine will be logged and denied. All other packets will be denied without being logged. In fact, this is a good time to define two new chains, perhaps called *logged-in* and *logged-out*. In that case, the rules would be as follows:

```
ipchains -A input -j logged-in -d 135.207.10.208
ipchains -A input -j logged-out -s 135.207.10.208
ipchains -A input -j DENY

ipchains -A logged-in -j DENY -p TCP -d 135.207.10.208 irc -l
ipchains -A logged-in -j DENY -p TCP -d 135.207.10.208 ssh -l
ipchains -A logged-in -j DENY -p TCP -d 135.207.10.208 telnet -l

ipchains -A logged-out -j DENY -p UDP -s 135.207.10.208 -l
```

This setup adds two new rules to the `input` chain, and then creates the `logged-in` and `logged-out` chains. These can be manipulated to log those services that you want to log. If disk space for logs is not an issue, then it is always best to log everything and then weed out the boring stuff later. It's a good idea to invest some time developing log processing scripts, and there are some good ones out there to be found.

DHCP introduces an interesting problem. The preceding example uses a particular IP address when rules are specified. In practice, *ipchains* commands are read in from files at start-up time. If the host is using DHCP to obtain an address, then there is no way to know in advance what the IP address will be. In that case, use a script with tools such as *grep*, *awk*, *sed*, and *perl* to discover its IP address, and then feed that value into the *ipchains* command in a script.

There may be a race condition here: Does the interface run briefly without rules after booting? And if the *ipchains* script fails, does it pass or suppress packets?

Ipchains has a nice feature that enables you to test the filtering once a set of rules is defined, using the "-C" option. For example, after the rules in the preceding example are entered, the command

```
ipchains -C input -p TCP -i eth0 -s 135.207.10.208 333 -d 207.140.168.155 www -y
```

tests to see if the machine can access the Web server on 207.140.168.155. Typing that in results in the output "accepted." However, the following command

```
ipchains -C input -p UDP -i eth0 -s 135.207.10.208 333 -d 207.140.168.155 www
```

results in the output "denied," as the rules do not allow arbitrary outbound UDP. These commands are useful, but relatively awkward.

11.3.2 Building a Firewall for an Organization

For the next example, we start with a minimally configured UNIX host—we used FreeBSD, but Linux, Sun, or almost any other would do. When deciding which operating system to use, it helps if you are familiar with administering the operating system, which should reduce errors. If you can afford it, use a dedicated machine, and turn off all services except those that are needed for the firewall to work. Secure the host using the guidance in Chapter 14.

We need an engine to install and execute our filtering rules. A number of filters are available, depending on the operating system. FreeBSD has *ipfw* and *ipf*. Ipchains is available on Linux. Apple's OS X (which is built on FreeBSD) also uses *ipfw*, and a GUI called *BrickHouse* is available, although we prefer the command line. OS/X.2 comes with a very restrictive GUI that enables you to block inbound ports, but does not do any filtering based on addresses, and there is no way to control outbound traffic. Fortunately, both the built-in GUI in Jaguar and BrickHouse are just front ends for *ipfw*, and once the rules are in place, you can still edit them manually from the console.

Ipfw runs in the kernel, and has a variety of options. It has stateful inspection, which keeps track of individual TCP sessions and only allows packets through that continue properly started connections—this is implemented with a dynamic ruleset. It supports dynamic address translation. Packets for particular destination hosts or services can be diverted to proxies, loggers, and so on. This can offload traffic that requires special handling. *Ipfw* also has *traffic shaping*, which can slow or even out the flow of packets for more consistent or controlled traffic. It can implement algorithms such as RED queue management [Braden *et al.*, 1998]. *Ipfw* also drops several kinds of pathological IP fragments that should never appear in innocent network traffic.

Ipf is a kernel-based packet filter written by Darren Reed. It has a readable configuration language with a well-defined syntax, including a BNF description. Oddly enough, both *ipf* and *ipfw* are available in the FreeBSD kernel, though they operate separately. By default, *ipf* examines all rules before processing a packet. One needs the "quick" keyword to invoke the more useful immediate processing, which tends to burden our configuration with extra text. "Quick" is a bad idea. It complicates rule execution order, and makes rulesets difficult to read. Put the "quick" statement on every line, and then pay attention to the order.

For this example, we examine the firewall rules actually used by a small company. They started with a commercial firewall, but found FreeBSD and *ipf* easier to install, administer, and understand. For simplicity, we extended *ipf* in an important way: We are using macros to name the various firewall interfaces, networks, and relevant hosts. *Ipf* does not have this naming capability, though many firewalls do, including many GUI-based ones. This naming is important: It makes the rules more understandable, and simplifies changes to the firewall ruleset. It is vital to document these rulesets, as it is likely that the original installer will have moved on when changes are needed.

Note that we did not actually change the *ipf* code itself. Instead, we used the familiar C preprocessor to do the work for us—one could also use the m4 macro processor.

First, we need to define the interfaces on the firewall. Much filtering is usually done based on the interface that is handling the traffic—in most cases, this gives us important topological information. For example, one interface probably connects directly to the router leading to the Internet. Incoming traffic on that interface is the most obviously suspect.

We had to make some compromises for the presentation of this example. First, the lines are too long for this book, so we've had to break the lines for readability. An actual `ipf.conf` file is easier to read without the line breaks. Second, this example is derived from the actual firewall rulesets of a small company, but it has been edited for clarity—we've removed some of their rules and special cases, and rearranged things. We've also tightened things up by adding rules from `ipf.conf.restrictive`, one of the sample files that comes with the *ipf* package. Books and papers should use tested programs and scripts, but that was not possible here, so our only guarantee of correctness is hand-checking.

Three networks are connected to this firewall: the Internet, a DMZ, and the inside network. The DMZ contains hosts to offer Web and DNS service to the Internet, and to provide mail and time (NTP) transport across the firewall.

We start with some definitions:

```
#define IF_INTERNET     fxp0
#define IF_INSIDE       fxp1
#define IF_DMZ          fxp2

#define INT_NET         xx.xx.xx.128/25
#define US              xx.xx.xx.0/24
#define DMZ_NET         xx.xx.xx.64/27

#define INT_SMTP1       xx.xx.xx.133
#define INT_SMTP2       xx.xx.xx.134
#define INT_NTP         xx.xx.xx.133
#define EXT_SMTP1       xx.xx.xx.66
#define EXT_SMTP2       xx.xx.xx.67
#define EXT_NTP         xx.xx.xx.67

#define GUARD           xx.xx.xx.131
#define FIREWALL        xx.xx.xx.5
#define WEBSERVER       xx.xx.xx.67     // in DMZ
#define LOGGER          xx.xx.xx.133

# protocol definitions
```

```
#define TRACEROUTE_RANGE 33434 >< 33690
#define SYSLOG          514
#define ICMP_PING       8
#define DNS_PORT        53
```

We have been assigned a single /24 network, xx.xx.xx.0/24, a.k.a. "US." A thirty-two host-range starting at .64 comprises our DMZ. The other hosts are at or inside our firewall. (This example does not use the other 96 possible addresses in US.) We define a few of the ports for readability, though we think that the distribution should include a file with all of these defined. Note that we specify hosts by the services they provide. xx.xx.xx.133 provides several services, but we give it different names, in case we have to move the services.

Next we set an environment for the rest of the rules. If we take care of spoofing problems here, it makes the remaining rules cleaner:

```
### first, some general rules
#
# Nasty packets which we don't want near us at all
# packets which are too short to be real.
block in log quick all with short
block in log quick all with opt lsrr
block in log quick all with opt ssrr

# loopback packets left unmolested
pass in  quick on lo0 all
pass out quick on lo0 all

# Drop incoming packets from networks that aren't routable
block  in     quick from 192.168.0.0/16 to any
block  in     quick from 172.16.0.0/12 to any
block  in     quick from 10.0.0.0/8 to any
block  in     quick from 127.0.0.0/8 to any
block  in     quick from 0.0.0.0/32 to any
block  in     quick from 224.0.0.0/3 to any

# Block incoming spoofs from the Internet
block  in     quick on IF_INTERNET from US to any

# we may not send spoofed packets,  nor multicast
block out log quick on IF_INTERNET from !US to any
block in  quick on IF_INTERNET to 224.0.0.0/3 to any
```

This is pretty much boilerplate, though you may want to allow multicast if your security policy permits it. There are other pathological packets that should probably be dropped. We log some of these packets, but an administrator may not care if someone on the Internet is sent weird packets. Conversely, it can be useful to know if you are under attack.

Next we set the rules for accessing the firewall itself. This firewall is running at network layer 3, *i.e.,* it has its own IP address. We want almost no one to be able to reach it. We do want *ssh* access to it from the internal network, but not from the outside:

```
### access to the firewall itself #####
# only insiders may ssh, ping, or traceroute to it.
```

```
pass  in log  quick on IF_INSIDE proto tcp from INT_NET to
                                 FIREWALL port = 22 flags S keep state
block in log  quick on IF_INTERNET       proto tcp from any to
                                 FIREWALL port = 22
pass  in      quick on IF_INSIDE proto icmp from INT_NET to
                                 FIREWALL icmp-type ICMP_PING keep state
pass  in      quick on IF_INSIDE proto udp  from INT_NET to
                                 FIREWALL port TRACEROUTE_RANGE keep state
```

The firewall can store its own logs, but it is also wise to send the log messages to a remote drop safe within the secured area:

```
# syslog drop safe for the firewall
pass  out     quick on IF_INSIDE proto udp port SYSLOG to LOGGER

# If any alien or unexpected program tries to access anywhere from the
# firewall, block and log it.
block out log quick on any from FIREWALL to any

# no other incoming access to the firewall
block in      quick on any       to FIREWALL
```

At this point, there are a couple of ways to arrange the rules. We can group all the rules for a particular network together, or we can group the rules by the services they implement. The former makes it easier to audit network use, the latter helps us understand how each service is implemented. We choose the latter, and we will describe some general rules about each of the networks.

We'll start with e-mail, which is transported by SMTP. There are e-mail relays in the DMZ, and on the inside network. Each has two machines, for robustness. We relay all incoming mail through the DMZ host to the internal mail relay, where it gets filtered for spam, viruses, and so on, and is forwarded to users. We let users and the internal mail relay send outgoing mail themselves. Some companies may insist on filtering outgoing mail as well (a very good way to see if your company is infected and a source of viruses!):

```
# Incoming e-mail from the Internet goes to our DMZ mail relay host.
pass  in     quick on IF_INTERNET proto tcp from !US to
                                      EXT_SMTP1 port = 25 keep state
pass  in     quick on IF_INTERNET proto tcp from !US to
                                      EXT_SMTP2 port = 25 keep state

# DMZ mailers then forward to internal servers
pass  in     quick on IF_DMZ      proto tcp from EXT_SMTP1 to
                                      INT_SMTP1 port = 25 keep state
pass  in     quick on IF_DMZ      proto tcp from EXT_SMTP1 to
                                      INT_SMTP2 port = 25 keep state
pass  in     quick on IF_DMZ      proto tcp from EXT_SMTP2 to
                                      INT_SMTP1 port = 25 keep state
pass  in     quick on IF_DMZ      proto tcp from EXT_SMTP2 to
                                      INT_SMTP2 port = 25 keep state

# Allow the inside mail relays to reach the DMZ hosts
pass in  quick on IF_INSIDE proto tcp from INT_SMTP1 to
```

```
                                        EXT_SMTP1 port = 25 keep state
pass in  quick on IF_INSIDE proto tcp from INT_SMTP1 to
                                        EXT_SMTP2 port = 25 keep state
pass in  quick on IF_INSIDE proto tcp from INT_SMTP2 to
                                        EXT_SMTP1 port = 25 keep state
pass in  quick on IF_INSIDE proto tcp from INT_SMTP2 to
                                        EXT_SMTP2 port = 25 keep state

# Note: many sites let the inside mailers deliver directly to internet
# destinations.  This rule forces them to go through the relays.
# Uncomment it if that's your policy.
# block in quick on IF_INSIDE proto tcp from US to any port = 25 keep state

# Finally, allow the DMZ relays to send mail into the world.
#
pass in  quick on IF_DMZ    proto tcp from EXT_SMTP1 to
                                        !US port = 25 keep state
pass in  quick on IF_DMZ    proto tcp from EXT_SMTP2 to
                                        !US port = 25 keep state
```

These examples make no provision for *smtps*, and they should. We should be encouraging encrypted transport, not blocking it.

Our support of the DNS protocol is quite similar to SMTP:

```
# incoming DNS queries
pass  in     quick on IF_INTERNET  proto tcp/udp from any to
                                      EXT_DNS1 port = DNS_PORT keep state
pass  in     quick on IF_INTERNET  proto tcp/udp from any to
                                      EXT_DNS2 port = DNS_PORT keep state

# our DMZ DNS servers can talk to the inside DNS relays:
# (we don't need to keep the bogus UDP "state" since these
# are simple bidirectional channels
pass  in     quick on IF_DMZ    proto tcp/udp from EXT_DNS1 to
                                      INT_DNS1   port = 53
pass  in     quick on IF_DMZ    proto tcp/udp from EXT_DNS1 to
                                      INT_DNS2   port = 53
pass  in     quick on IF_DMZ    proto tcp/udp from EXT_DNS2 to
                                      INT_DNS1   port = 53
pass  in     quick on IF_DMZ    proto tcp/udp from EXT_DNS2 to
                                      INT_DNS2   port = 53

# inside DNS hosts can talk back to DMZ servers
pass  in     quick on IF_INSIDE proto tcp/udp from INT_DNS1 to
                                      EXT_DNS1 port = 53
pass  in     quick on IF_INSIDE proto tcp/udp from INT_DNS1 to
                                      EXT_DNS2 port = 53
pass  in     quick on IF_INSIDE proto tcp/udp from INT_DNS2 to
                                      EXT_DNS1 port = 53
pass  in     quick on IF_INSIDE proto tcp/udp from INT_DNS2 to
                                      EXT_DNS2 port = 53

# outgoing DNS queries from the DMZ
pass  in     quick on IF_DMZ      proto tcp/udp from EXT_DNS1 to
```

```
                                        !US port = 53 keep state
pass  in     quick on IF_DMZ      proto tcp/udp from EXT_DNS2 to
                                        !US port = 53 keep state
```

The hosts INT_DNS1 and INT_DNS2 should filter DNS responses, not just relay them. People can inject nasty DNS responses.

NTP traffic is about the same, but with no redundant hosts:

```
# NTP traffic from the world into us...
pass  in     quick on IF_INTERNET proto udp port = 123 from any to
                                        EXT_NTP
pass  in     quick on IF_DMZ      proto udp port = 123 from EXT_NTP to
                                        INT_NTP

# ... and back out
pass  in     quick on IF_INSIDE    proto udp port = 123 from INT_NTP to
                                        EXT_NTP
pass  in     quick on IF_DMZ       proto udp port = 123 from EXT_NTP to
                                        any
```

There is one more major service, which we probably should have put earlier in the file for efficiency reasons, as it is likely to be busy:

```
# incoming Web queries
pass  in  quick proto tcp from any to
                        WEBSERVER port = 80 flags S keep state
pass  in  quick proto tcp from any to
                        WEBSERVER port = 443 flags S keep state
```

At this point, the remaining services are mainly based on the network leg. First, close shop on the DMZ:

```
# insiders can access ssh on the DMZ
pass  in log quick on IF_INSIDE proto tcp  from INT_NET to
                        DMZ_NET port = 22 flags S keep state

# logging drop-safe for DMZ hosts
pass  in     quick on IF_DMZ    proto udp port SYSLOG from DMZ_NET to
                        LOGGER

# all other traffic from DMZ is unexpected.  Have we been hacked?
block out log quick on IF_DMZ to all from all
```

For our inside users:

```
# Allow other arbitrary internal TCP access to the outside
pass in quick on IF_INSIDE proto tcp from INT_NET to
                        any flags S keep state

# permit ping to the Internet.  State code permits the pong as well.
pass in quick on IF_INSIDE proto icmp from INT_NET to
                        any icmp-type ICMP_PING keep state

# traceroute
pass in  quick on IF_INSIDE proto udp from INT_NET to
                        any port TRACEROUTE_RANGE keep state
```

Some final fun, and then the always wise final filter:

```
# Annoy anyone that tries to scan port SMTP or IDENT:

block return-rst in  quick on IF_INTERNET proto tcp from any to
                          any port = 25
block return-rst in  quick on IF_INTERNET proto tcp from any to
                          any port = 113

block in all
```

Ipftest

Ipf comes with a utility, *ipftest*, that can be used to check how a particular set of rules would handle traffic, without actually subjecting a network to that traffic. Data can be passed to *ipftest* from a raw file, or the output of *tcpdump* can be passed to a set of filter rules. The output of the program will be `pass`, `block`, or `nomatch`. A convenient feature is to take tcpdump output, edit it by hand for a situation that you want to test, and then run it through *ipftest* to see what happens. It is a very convenient program to use while designing a firewall.

Of course, there are other tools for testing a firewall as well. Run *netstat -a* if you have login access to the firewall, and *nmap* if you don't.

11.3.3 Application-Based Filtering

The previous examples dealt with packet-based filtering. On a host, it is possible to also filter based on applications. For example, on a Windows machine, users can specify that Internet Explorer is allowed to access the Internet, but Quicken is not. *Zonealarm* is an example of a program that gives users the ability to monitor and control the access of applications to the Internet. For each application, users can specify the addresses and ports that will or will not be blocked.

One of the challenges to application-based filtering is that it is not always clear what is meant by a *program*. If a worm does a DNS lookup, the query to port 53 may come from the machine's resolver, not the worm. It can't be blocked, but it should be. Is a Web browser's Java interpreter or integrated mailer part of the same "program" or not?

System programs tend to have obscure functionality and requirements, as far as the user is concerned. What decision should a user make if something called *IEFBR14.DLL* tries to access the Internet? If the user does not permit the access, and checks the little box to remember that decision and not be asked again, what things will break two weeks later? Will the user be able to associate that break with this decision? If the user allows the access, what dangers does he or she face?

Application-based filtering can be a good idea. It can do a better job containing worms than most traditional firewalls do, but design is critical. At a minimum, one should come with help features that provide additional information to users when the program complains about an application trying to access the Internet. And, of course, a great deal of care must be taken to ensure that the malware doesn't spoof the informational pop-up. ("EvilBackDoor.exe is a standard part of your Web browser, and comes pre-installed by government regulation on all operating

systems, including Windows, Solaris, and PalmOS. Do not, under any circumstances, block it from accessing the Internet as a server, or orange smoke will come out of your monitor.")

Be aware that some malware now seeks out and disables host-resident firewalls and virus filters.

11.4 Firewall Problems

Some problems arise by accident or simply out of negligence. Others are inherent problems. Firewalls interfere with many things users want to do, so enterprising users find ways around them and sometimes introduce new vulnerabilities.

11.4.1 Inadvertent Problems

"You have attributed conditions to villainy that simply result from stupidity."

Logic of Empire [1967]
—ROBERT A. HEINLEIN

Never ascribe to malice what can be adequately explained by incompetence.

Murphy's Law Book Two [Bloch, 1979]
—HANLON'S RAZOR

Some problems arise without any malicious intent on the part of users or administrators. For example, companies may institute a policy dropping all e-mail coming through the gateway, to avoid exposure to mail-borne viruses. However, if port 80 is left open, Web mail services introduce a new avenue for malicious code to get in, via e-mail-over-Web tunnels. People using services such as Hotmail in such an environment are guilty of violating policy, but not of being hackers. (There is a saying that "sometimes, the light at the end of the tunnel is the oncoming train." All manner of bad things can travel through your tunnels; see Section 12.1.)

Administrative errors are the most common cause of firewall problems. Very large rulesets, changes in personnel, legacy rules that do not change, and lack of documentation all make it difficult to manage firewalls. An administrator who inherits a firewall with poor documentation about the ruleset does not know if it is okay to remove a rule, or the effect that adding new rules will have. Rulesets tend to be unwieldy; the complexity of the policy that the firewall implements is often greater than the policy specified for the site.

In one case we're familiar with, a data center allowed each of its customers to specify five firewall rules to be added to the global ruleset. Customers can also purchase more rules. With firewall rules specified by different parties, how can they possibly have a coherent site policy?

11.4.2 Intentional Subversions

Long round trip time but hell of a good MTU.

On implementing NFS over IP over e-mail
—MARCUS RANUM

Some firewall problems are due to conscious efforts to subvert them. These can be due to users who want more functionality than the firewall offers, or to malicious parties attempting to subvert the site. For example, many firewalls are set up to maintain state about *ftp* connections. When an internal client issues an *ftp PORT* command to an external server, the firewall stores the port number for the data connection and allows the return connection through. Before the problem was fixed, some commercial firewalls allowed a *PORT* command, originating on the inside, to specify ports such as 23, which allowed someone on the outside to telnet directly to an internal machine [Martin *et al.*, 1997]. This attack, implemented as a JAVA applet, could enable an external party to open holes in the firewall on arbitrary ports. Vendors are now aware of this problem and have closed off low-numbered ports. (Of course, as we've pointed out, sensitive services may live on high-numbered ports.)

SOAP (see Chapter 12) can be used to transmit arbitrary protocols over HTTP. Firewalls often allow traffic destined for port 80 to pass, which is wrong. Inbound HTTP traffic should be allowed only to a Web server, and should not reach other internal machines. Besides, your externally accessible Web servers should be on a DMZ network. Gaynor and Bradner [2001] jokingly describe the *Firewall Enhancement Protocol (FEP)*, which is designed to overcome the communication obstacles presented by firewalls. But it's not just an April 1 RFC; *Httptunnel*[1] is a publicly available tool for transporting IP packets via HTTP.

Occasionally, someone who should know better pokes a "temporary" hole in a firewall to accomplish something or other. Often, the person then forgets about it, despite the fact that this is a deliberate violation that can cause a security problem.

Problems are also caused by systems that are designed to straddle the firewall. An internal and external proxy can maintain a control connection between them, and pass agreed-upon traffic [Gilmore *et al.*, 1999]. There is little an administrator can do about such a circumvention of the firewall. However, such systems are very useful, and the benefit of using them may outweigh the risk. Security is about managing risk, not banning it [Schneier, 2000].

11.4.3 Handling IP Fragments

The existence of IP fragmentation makes life difficult for packet filters. It is possible that the ACK or SYN bits in a TCP packet could end up in a different fragment from the port number. In fact, there are tools designed to break packets up in just that way. In these situations, a firewall cannot know if it should let something through, because it does not know if it is part of an existing

1. See `http://www.nocrew.org/software/httptunnel.html`.

conversation. 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.

The problem is triggered because of tiny initial fragments. These have no rational reason for existing. A simple way to avoid this problem is to require the initial fragment to be at least 16 bytes long. In fact, it is even better if it is long enough to cover an entire TCP header in case other options need to be looked at.

One could also reassemble fragments at the firewall, and a lot of firewalls do this. Errors in fragmentation processing can be a weakness in the firewall, though.

Assuming that initial fragments are long enough, if the main threat is penetration attempts from the outside, these 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 non-initial fragments and converting them back to proper packets on some outside machine.

You can do better if your filter keeps some context, even without doing reassembly. Mogul's *screend* [Mogul, 1989] caches the disposition of salient portions of the header for any initial fragment; subsequent pieces of the same packet will share its fate.

11.4.4 The FTP Problem

The FTP protocol has been a perennial problem for firewall designers. The firewall must open an FTP data channel in either direction based on commands in the control channel. If this is handled by something like a user-level proxy, it can be fairly straightforward. Care must be taken to ensure that the hole opened is appropriate, connecting the right endpoints, and vanishing if the control connection goes away. Furthermore, the control connection shouldn't time out if there is a long transfer.

But FTP is important enough, and seems easy enough, that may firewall designers attempt to implement the FTP transport in the kernel of the firewall at the packet level. This leaves them with the job of analyzing the control channel commands at the packet level. There are a number of tricks involving fragmentation and the like that make this job quite hard to get right.

It is an instructive test case to learn how a particular firewall handles FTP. One can get an insight into the security stance of the designers, if information is available at this detailed level.

11.4.5 Firewalking

Firewalls are designed to partition networks so that hosts on the outside cannot access internal hosts, except through a few authorized, and generally authenticated, channels. In practice, these channels often include some apparently innocuous but unauthenticated protocols. Thus, some firewalls allow ICMP echo and ICMP echo responses. Others allow DNS queries. However, these seemingly innocuous packets can actually be used to map hosts behind a firewall. The technique is called *firewalking* [Goldsmith and Schiffman, 1998].

There are a number of ways to do this. One way to firewalk is to add an option to the *traceroute* program that forces use of either ICMP or UDP packets, depending on which protocol is allowed through. Consecutive queries can be mounted while decrementing the TTL field to calculate the number of hops to a firewall. Then, the port number can be manipulated so that when UDP packets reach the firewall, their port number corresponds to the service that the firewall allows, such as port 53 for DNS queries. *Traceroute* can be further modified so that port number incrementing stops when the target port number is reached, thus permitting packets to be sent further past the firewall. In this manner, an attacker can guess IP addresses behind a firewall and probe to see if they exist and are up. In fact, this technique can be used to see not only if those hosts are up, but if they are running particular services. Source code for *Firewalk* can be found on the Net at `http://www.packetfactory.net/Projects/Firewalk/`.

11.4.6 Administration

We have seen institutions with 90–200 traditional, front-door firewalls. How long does it take to administer such a complex network? It is difficult to make sense of an organization like this. It is not likely that a consistent sitewide policy exists, nor that so many firewalls can be kept up-to-date when policy changes are required. It is hard to understand how such a configuration is possible, and yet it is not uncommon to find this many firewalls in large enterprises. How many people are in charge of 200 firewalls? One person cannot conceivably manage that many. If it is a group of people, then how are they keeping the policies coherent? If you have that many firewalls in your organization, you better have a real justification for it, and we can't think of any.

Some of those firewalls may be internal ones, providing extra security for sensitive areas. But we've also seen administrators wince upon hearing "but that firewall doesn't go to the outside." It may or may not—are you sure of your answer?

Another administrative problem arises from overlapping security domains. In a large organization, there are potentially many paths between any node and the outside. It is not always obvious what set of rules, or which failure of an application, leads to a particular point being exposed. If a user modifies his or her kernel so that it sends a packet out of one interface, with a return address on another interface, it may be possible to violate a security policy. For example, the user could *telnet* through another part of the organization. Application-level gateways can be less vulnerable to this because there is no IP connectivity. Rather, there should be no IP connectivity; if your network is too large, are you sure?

11.5 Testing Firewalls

Testing a firewall is not fundamentally different from testing any other piece of software. The process can determine the presence of bugs, but not their absence. When testing software in general, two common techniques are black box testing and white box testing. The former assumes no knowledge about the internals of the software and tests its behavior with respect to the specification and many different inputs. The latter utilizes knowledge of the code to test how internal

state responds to various inputs. Both mechanisms should be used when testing a firewall. It is important to inspect rules both manually and in an automated fashion. At the same time, it is valuable to bang against the firewall with real data.

Once you've built a decent test script, keep it and rerun it any time your configuration or your software changes. This sort of regression testing can catch many failures.

11.5.1 Tiger Teams

Tiger teams attempt to stress-test a firewall by mounting actual probes against it. The politics of letting loose a tiger team within an organization are addressed in Section 6.9. Here, we look at the technical aspects.

The most important thing to do before deploying a tiger team is to define the rules of engagement. What is the team allowed and not allowed to do? Is dumpster diving okay? What about social engineering [Mitnick *et al.*, 2002; Winkler and Dealy, 1995]? The nontechnical adjuncts to firewall testing are often the most likely avenues of actual attack. You want to find professionals. Anyone can take off-the-shelf tools and run them against your network. If you do it yourself, you may not know about some tools such as *fragrouter* [Song *et al.*, 1999], which is designed to evade naïve firewalls (see Chapter 15). Hire reputable people who tiger team for a living.

An example of something you do not want to permit in the rules of engagement is changing a domain name registration to point to another site. This causes damage to the site being tested. At the same time, hackers are not limited by these sorts of rules. One possibility is to duplicate a site with a different domain name, and then run the tests against the duplicate. Doing that, the tiger teams can operate under fewer restrictions. However, even the slightest difference between the test version of the site and the actual production site can result in an overlooked vulnerability. Do you know *all* salient details of the configuration, including such things as the protection mechanism specified for your domain name registration?

It is a good idea to run tiger teams on a regular basis because network architecture, firewall rules, and software environments change. General Curtis Lemay is quoted as having said that the *Strategic Air Command (SAC)* should be "a peacetime air force on a wartime footing." His tiger teams had the goal of leaving an empty beer can in the cockpit of a B-52. If they succeeded, someone was in big trouble. Everybody knew that it could happen at any time. Similarly, if network operators and administrators believe that tiger teams are testing their networks and firewalls at any given time, they will be much more diligent. But note that there are two different failure modes: those that occur because the sysadmins are asleep at the switch, and those that reflect actual technical failings. Both need to be fixed, but the fixes are different.

It is important to define the outcomes. For example, is it a success or a failure if the attackers do not get in but the attack is not noticed? There are also different levels of "getting in." With a defense in depth strategy, perhaps the attack gets through some layers but not others. That *is* a failure, as it shows that some level of defense didn't do its job.

11.5.2 Rule Inspection

The Rules

The number of rules is the best indicator of the complexity of a firewall. If you have 10 rules, perhaps you can analyze it; if you have 250 rules, why do you have so many? Perhaps a series of administrators managed the firewall and each was afraid to undo something the other did. We've analyzed firewall rulesets for clients. A number of times, after viewing a large, broken set of rules, we commented, "Surely, this is an internal firewall." The looks on the faces of the clients at that moment seemed to contradict this observation.

If you are using a version management system such as *cvs* to manage firewall rules, changes are logged and annotated. This leaves some hope for a coherent story about how the active firewall ruleset was derived. (By the way, can your favorite GUI do this?) Watch out for "temporary" changes to the rules. Often, they remain in place longer than expected. This is another reason to retest the firewall rules regularly. Perhaps rules should have an optional expiration date.

It is important to test the obvious as well as the non-obvious. When testing a prominent Web server once, as a favor, we happened to try *telnet*, and low and behold, it worked! It took them three tries to fix it.

If you have multiple firewalls, test from different places on the outside to make sure the rules are consistent. Different firewalls may have different rules, but you may not observe this if traffic is going through only one of them. If you have a fail-over, such as a hot spare configuration, then fail one and see not only if the other one works, but if it is doing the right thing.

Manual Inspection

Manually inspecting the firewall rules is important. Just as code walk-throughs reveal unintended mistakes and are an integral part of testing, reading through the rules by hand and justifying each one is a necessary part of testing. You may find that a "temporary" rule was not removed, or that you no longer understand why a particular rule exists. This is where the value of annotations is most noticed.

However, there is a limit to the amount of testing that can be done manually. Any firewall with a multitude of rules is too complex to analyze in your head, and thus manual inspection is a necessary but not sufficient exercise for analyzing the firewall.

Computer-Assisted Inspection

When testing the rules, build regression tests, write scripts, and test both by IP address and host name. It is important to test against every rule. There are also issues of interaction of the rules that can open up things you do not want opened. Chapman [1992] shows how difficult it is to set up secure rules for a packet filter. When testing, look for rules with wildcards; these are more likely to get you in trouble. In addition, look for rules that partially overlap each other.

[Mayer *et al.*, 2000] describes a tool for analyzing firewall configurations. It's a good start, but it's a supplement for testing, not a replacement.