Linux firewall facilities for kernel-level packet screening

Jos Vos <jos@xos.nl>
Willy Konijnenberg <willy@xos.nl>

X/OS Experts in Open Systems BV
Kruislaan 419
1098 VA Amsterdam
The Netherlands

Last updated: November 18, 1996

Abstract

The freely available Linux operating system includes a number of facilities for efficient kernel-level IP packet filtering and screening. The acceptance and forwarding of IP packets can be regulated by specifying filter rules, using packet and network device characteristics, such as IP addresses, port numbers, IP flags, and incoming or outgoing interfaces. Linux also provides support for transparent proxy servers by means of packet redirection, which is implemented as part of the IP firewall module and can be configured using a similar set of rules. Transparent here means that use of such a proxy server does not require any changes for users or application software. Finally, Linux can masquerade forwarded packets, so that it looks like all packets come from the Linux host.

1 Introduction

Network security, and more specifically the use of *Internet firewalls*, is one of today's hottest topics in the computer business. Every private network that is going to be connected to the Internet needs an appropriate firewall, being some combination of hardware, software, and procedures, to protect it. Most commercial firewall products are quite expensive, especially for small companies.

An alternative is to use *Linux*, a freely available operating system. We will mainly focus on one aspect of Linux, the IP packet screening facilities, being one of the components for building firewalls based on Linux. The Linux

packet screening facilities also provide a mechanism to support transparent proxy servers, which will also be described. At the end you'll find some recommendations for using Linux systems as a complete firewall solution.

This paper is based on release 2.0.25 of the Linux kernel and version 2.3.1 of the *ipfwadm* utility. Be aware of the fact that some details might change in future releases of Linux.

2 IP packet filtering

Before describing the Linux implementation of IP packet filtering, we will briefly introduce the underlying general concepts.

IP packet filters inspect network datagrams (IP packets) and decide whether these packets are allowed to pass the filter or not. These inspections may take place at several stages, like the moment packets arrive on the system or when they are about to be forwarded to another system.

The decision to let a filter block certain packets is usually based on several criteria, being checked against the contents of the IP packet and some environmental parameters:

- Source and destination IP addresses. This allows an administrator to restrict the traffic through the system by allowing only packets coming from (or sent to) a set of well-known hosts or networks. These addresses can be found in the header of every IP packet.
- Protocol, like *TCP*, *UDP*, or *ICMP*, which can also be found in the IP header.
- IP options. One of the more well-known options is the *source route* option, which can be considered to be dangerous in itself. Packets carrying this option are often refused.
- Source and destination port numbers, associated with TCP or UDP services. These port numbers can be found in the TCP and UDP header. Filters using port numbers can restrict the network traffic to a limited set of services, each of which is associated with a well-known port.
- TCP flags, being part of the TCP header. Most packet filters allow the TCP *ACK* and/or *SYN* bits to be checked, indicating whether a new connection is being set up or not. This is particularly useful when you want to allow some TCP sessions being initiated only from one side (e.g., only from hosts in your own network).

- ICMP message types, being defined in ICMP packets. This enables you to restrict ICMP traffic to a limited set of message types. You may, for example, want to refuse *Echo Request* packets (as used by the *ping* utility).
- User data, found in the (protocol-specific) data part of the IP packet. This is a tricky one, which normally doesn't belong in IP packet filters, although in some cases its use can't be avoided.
- Network device, via which a packet is received or is going to be sent. This enables you to let a filter act differently for different network devices (like external and internal interfaces).
- Date and time. This can be used to limit some types of network traffic to office hours, for example.

Some of these criteria are easy to check, such as the ones being part of the IP header. Others may be more difficult to handle, such as the information found in the TCP/UDP headers. One IP packet, for example, may be split up in two or more IP fragments, of which only the first fragment contains the TCP header. Furthermore, IP packets may not pass the filter in sequence, because session control is handled in the TCP layer. Some packet filters address some of these problems, others don't.

An IP packet filter can react in different ways to packets, depending on the filters and the capabilities of the filtering system. The most obvious actions are: let the packet pass the filter normally or drop the packet silently. Some filters also offer the possibility to send some notification (like an *ICMP Destination Unreachable* message) back to the sender, in case a packet is blocked by the filter. Besides that, filters often include some logging facility, to facilitate the network administrator in detecting if someone is trying to break in.

There are several ways to implement packet filters in a UNIX system. The most efficient way is to implement it in the kernel. One of the disadvantages of this method is that it is not very flexible. Another method is to let some user-level process do the filtering. Such a daemon process has to get the relevant information of all IP packets (at least part of the IP and TCP/UDP headers, possibly even the data part) via some system call, apply its filter rules, and pass the answer back to the kernel. The (freely available) *screend* package behaves like this. The major drawback of these implementations is the considerable overhead generated by the system calls and process scheduling.

3 Kernel filtering scheme

The Linux kernel provides native support for IP packet filtering at several stages: when a packet is received, when packet is sent, and when a packet is being forwarded (see figure 1).

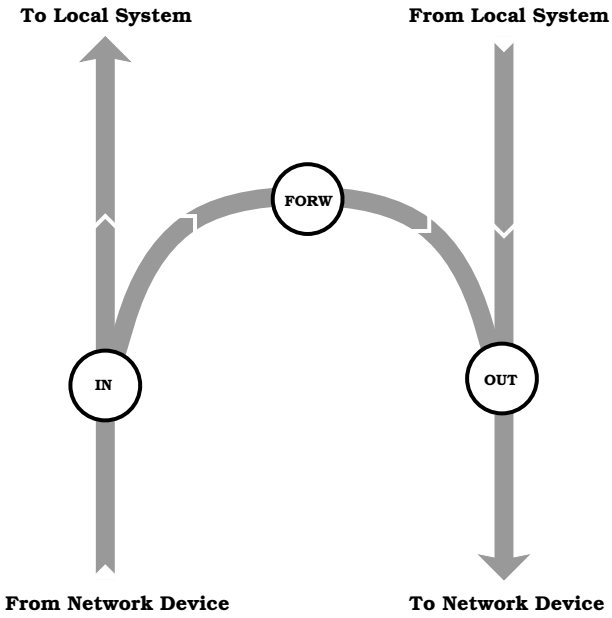

Figure 1

Each of the three filters consists of a default policy and a list of filter rules. Every filter rule defines some packet characteristics, like IP addresses, an optional network device, and several other options. Furthermore, each rule has a policy associated with it, defining what to do when a packet matches with the rule.

The algorithm used in the filters can be described as follows:

1. Step through the list of filter rules associated with the filter and check whether the packet matches with the rule or not.
2. The first matching filter rule (if any) determines all further actions:
   - The rule's policy will be applied to the packet.
   - Each rule contains packet and byte counters, which will be incremented when a packet matches.
   - Optionally, some information about the packet is written to the Linux kernel log.

- Finally, a rule may contain parameters defining how to change the *TOS-field* in the IP header, dealing with the packet's priority.

3. If none of the filter rules match with the packet, use the default policy associated with the filter.

There are currently three policies supported in Linux:

- *Accept*: let the packet pass the filter.
- *Deny*: silenty drop the packet.
- *Reject*: drop the packet and send an ICMP Destination Unreachable message back to the sender as a notification.

Filter rules in Linux contain the following items:

- IP source and destination addresses, both with their own 32-bit mask. Although the most common use of the mask is some regular netmask, to let a rule cover a complete (sub)network, the mask may contain an arbitrary bit pattern. A mask containing only 0's matches with every address.
- The protocol, being either *TCP*, *UDP*, *ICMP*, or "any".
- Source and destination port numbers (services), used in combination with TCP or UDP packets. Up to ten source and destination ports may be specified in one rule, including ranges of ports (like 1024-65535, representing all unpriviliged ports).
- Message types, used with ICMP packets.
- Bits to match with the TCP flags *ACK* and *SYN*, used to refuse the establishment of new TCP connections in a certain direction.
- The name or IP address of a network device. Rules containing a device specification will only match with packets coming in (or going out) via that particular device.
- A specification for changing the *TOS-field* in the IP header, being used when a packet is accepted by that rule.
- A flag indicating if some basic packet information should be written to the Linux kernel log, in case a packet matches with that rule.

Now that we have seen the basic concepts of the Linux firewall filters, we will show how to manage the filter rules from an administrator point of view.

4 Managing firewall filters

The interface for managing the kernel-level filter rules at user-level mainly consists of two parts:

- Firewall rules can be created or deleted via the *setsockopt*(2) system call. Via the same mechanism the default policy of a filter can be changed. All details of this interface are descibed in the *ipfw*(4) manual page.
- Filters can be inspected by reading the following pseudo-files in the */proc* filesystem:

  ```
  /proc/net/ip_input
  /proc/net/ip_output
  /proc/net/ip_forward
  ```

  Each of these files lists the default policy, followed by the details of all rules (if any) belonging to that filter, in a compact format.

The *ipfwadm* command provides a command-level interface for managing the Linux firewall facilities: it can be used to change or inspect all aspects of the kernel filters. Let's start with a simple example:

```
ipfwadm -I -a deny -S 192.168.22.0/24 -D 0.0.0.0/0
```

This command basically means "refuse all incoming packets originally coming from network 192.168.22.0". It appends (`-a`) a new rule to the list of filter rules belonging to the *input* firewall (`-I`). The *output* and *forward* filters can be changed by using the `-O` and `-F` options, respectively. After the append command the rule's policy is specified. Valid keywords are `accept`, `deny`, and `reject` (refuse the packet, but return an ICMP message). The source (`-S`) and destination (`-D`) addresses both include a mask: the suffixes `/24` and `/0` are equivalent to `/255.255.255.0` and `/0.0.0.0`, respectively. The following commands are equivalent with the above command:

```
ipfwadm -I -a deny -S 192.168.22.0/24 -D 11.22.33.44/0
ipfwadm -I -a deny -S 192.168.22.0/24 -D any/0
ipfwadm -I -a deny -S 192.168.22.0/24
```

Every IP address will match the specified destination, because the mask only contains 0's. Also, the specified name will not be resolved when a zero-mask is specified, so `any/0` or `somehost/0` will also work. Finally, the default

source and destination addresses are "anywhere", so the `-D` option may be omitted in the above command.

Another example:

```
ipfwadm -I -a accept -k -P tcp -S any/0 telnet \
        -D 192.168.37.1 1024:65535
ipfwadm -O -a accept -P tcp -S 192.168.37.1 1024:65535 \
        -D any/0 telnet
```

The command creates two rules (one for the input firewall, one for the output firewall) that accepts all packets belonging to an outgoing telnet connection (here it is assumed that our local IP address is 192.168.37.1). The protocol is specified via the `-P` option and after the IP address a service name (`telnet`, specifying port 23) and a port range (in this case all unpriviliged ports) are specified. The `-k` option makes the input rule only match with packets having the TCP ACK flag set. This prevents someone from trying to initiate a connection from the outside (using source port 23) to some unpriviliged port on our system. Note that the `any/0` specification may not be omitted in these commands, because a specific port is specified.

Unfortunately, enabling a service is not as easy for all services. The *ftp* protocol, for example, uses a separate, incoming connection to transfer the data. So, using ftp (unless used in "passive mode") requires to allow a connection being initiated from outside your own network. Setting up a set of firewall rules for ftp would look like:

```
ipfwadm -I -a accept -k -P tcp -S any/0 ftp \
        -D 192.168.37.1 1024:65535
ipfwadm -O -a accept -P tcp -S 192.168.37.1 1024:65535 \
        -D any/0 ftp
ipfwadm -I -a accept -P tcp -S any/0 ftp-data \
        -D 192.168.37.1 1024:65535
ipfwadm -O -a accept -k -P tcp -S 192.168.37.1 1024:65535 \
        -D any/0 ftp-data
```

Here it is assumed that `ftp-data` is a valid service name for TCP port 20.

With the `-p` (set policy) command a default policy is specified.

```
ipfwadm -F -p deny
```

In this case, all forwarding is disabled, unless packets match with one of the forward rules, explicitly allowing them to pass the filter. Filters can be listed using the `-l` command, like in:

```
ipfwadm -I -l
```

This command would produce the following result, after issuing the above *ipfwadm* commands for the input firewall:

```
IP firewall input rules, default policy: accept
type  prot source            destination       ports
deny  all  192.168.22.0/24    anywhere          n/a
acc   tcp  anywhere           gw.foo.com        telnet -> 1024:65535
acc   tcp  anywhere           gw.foo.com        ftp -> 1024:65535
acc   tcp  anywhere           gw.foo.com        ftp-data -> 1024:65535
```

The printed hostname, `gw.foo.com`, corresponds to the local system, having IP address 192.168.37.1. There are several options to change or extend the given output of *ipfwadm*. The above example shows the most simple format.

Some more hints for managing firewall filters:

- The order of the filter rules is important: only the first matching rule is taken into account, so the *ipfwadm* commands should be given in the correct order.
- Combine rules (by specifying multiple port numbers or service names) as much as possible, because checking filter rules for every IP packet uses some CPU-time.
- Be sure to define the filters at the appropriate time when starting up the system. The best place to do this is *before* the network devices are configured (using the *ifconfig* command).
- Although *ipfwadm* allows you to specify host or network names when defining filter rules, this will not work in most cases, because the system probably can't resolve names before the network is operational. For the same reason, use the `-n` option (numeric output) when listing filters on moments the network is not (yet) operational.
- When you need to change a filter on an operational system, issue the necessary commands in the right order:

  1. Set the default policy of the filter to `deny`.
  2. Remove all rules belonging to this filter (*ipfwadm*'s `-f` option).
  3. Set up the new set of filter rules.
  4. Set the default policy to the desired value.

  This ensures that you don't have any time intervals, during which network traffic is not controlled by the firewall.

See the *ipfwadm*(8) manual page for more details and other options.

5 Transparent proxying

A complete firewall solution usually not only includes IP packet filters, but also some type of application-level proxy servers. Unfortunately, many proxying techniques need software modifications at the client side (think of *SOCKS*) or change the user-interface, requiring users to become proxy-aware. For this reason, the concept of *transparent proxying* was introduced in some of the more modern firewalls. This not only relies on special user-level software (modified proxy servers), but it also requires kernel-level support, which is now included in Linux. Transparent proxying redirects sessions passing the firewall to local proxy servers in a fully transparent way. Clients (both software and users) do not know their session is handed over to a proxy process: they still think they have a direct connection with the target they specified. Because it relies on port numbers, transparent proxying only works for TCP or UDP traffic.

In the Linux kernel, packet redirection for transparent proxying is mainly handled by the input firewall. An introductory example:

```
ipfwadm -I -a accept -r 2323 -P tcp -S 192.168.37.0/24 \
        -D any/0 telnet
```

This command redirects (`-r` option) all telnet sessions originating from the 192.168.37.0 network to a local (proxy) telnet server, listening on port 2323. When no port (or port `0`) is specified with the `-r` option, the port number of the original destination will be used as target port on the local host:

```
ipfwadm -I -a accept -r -P tcp -S 192.168.37.0/24 \
        -D any/0 smtp www gopher z3950
```

Now all incoming sessions using one of the specified protocols will be redirected to servers on the local host, using the original port number. These servers usually have to be specially prepared proxy servers. We will not discuss the general concept of proxy servers here, but only introduce the techniques available for changing existing proxies intro transparent ones. For TCP sessions, these techniques include:

- The original destination (IP address and port number) can be requested via the *getsockname*(2) system call. The returned *sockaddr* structure will contain information about the originally specified target, not about the local host.
- Packets sent back via the accepted socket will automatically carry the original destination address and port as sender address. So, the client will think the packet comes from the original target.

- A server may (given it has appropriate privileges) also explicitly bind a socket to a port with an external IP-address. This will cause packets for that address (and port) to be redirected to that local server, without a separate firewall rule.

For UDP, the situation is a bit more complex, especially because the *getsockname* system call can not be used:

- For use in UDP proxy servers, two macros have been defined for sending and receiving proxied UDP packets. They can be found in the file `/usr/include/sys/socketproxy.h`.
- The macro *recvfromto* is essentially equal to the *recvfrom*(2) system call, except that it takes two extra parameters of type `(struct sockaddr *)` and `(int *)`, respectively. In this second sockaddr structure, the original destination of the packet will be stored, so the proxy server knows where the packet was sent to.
- In a similar way, the macro *sendtofrom* is a wrapper for the *sendto*(2) system call, also having two extra parameters of type `(struct sockaddr *)` and `(int *)`, respectively. In this second sockaddr structure, the desired (fake) source address can be given, so the client thinks the packet is coming from there.

With these few interfaces, it is fairly simple to modify an existing proxy server to be transparent.

Note that redirecting packets to other systems requires user-level software. Such a function can be implemented by redirecting the packets to a local server acting as a simple packet-forwarder.

6 Masquerading

The Linux kernel provides an additional mechanism to use in firewall solutions: masquerading of IP packets. This means that some or all packets being forwarded by a Linux system can be changed as if there were sent from the local system. So, the source IP address is replaced by the local IP address and the source port is replaced by a locally generated port (e.g., 60005). Because an administration is kept of masqueraded sessions, incoming packets for that port will automatically be "demasqueraded" and forwarded to the system that originally initiated the session.

The next table summarizes the masquerading function, given a telnet session from an internal host (192.168.37.15) to an external host (10.42.17.8), passing a Linux system doing masquerading (192.168.37.1):

	source		*destination*	
	IP address	*port*	*IP address*	*port*
→ original packet	192.168.37.15	1027	10.42.17.8	23
→ masqueraded	192.168.37.1	60005	10.42.17.8	23
← reply packet	10.42.17.8	23	192.168.37.1	60005
← demasqueraded	10.42.17.8	23	192.168.37.15	1027

Masquerading takes place after passing the forward firewall filter. Demasquerading is done after receiving a packet and demasqueraded packets bypass the forwarding filter. Figure 2 shows the kernel flow diagram including (de)masquerading.

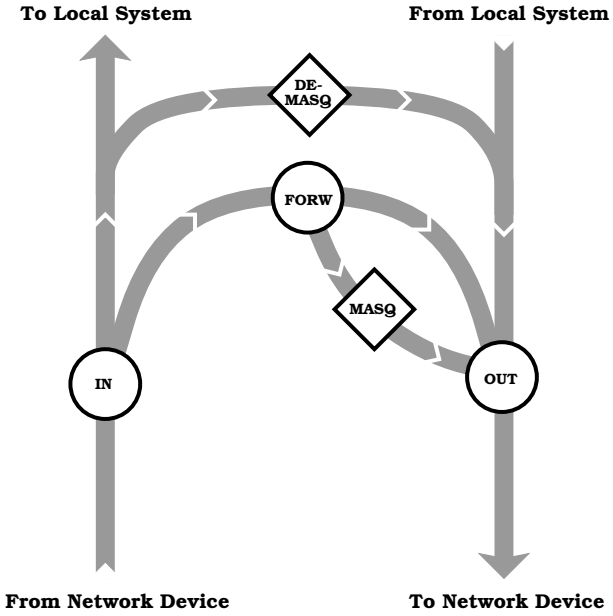

Figure 2

Masquerading is not as easy as it seems: some protocols need special care. One of the problem areas is found in the widely used *ftp* protocol, because this protocol uses a second session (normally initiated by the remote site) for transferring the actual data. A similar problem arises with some other popular protocols, like *IRC* and *RealAudio*. The Linux IP masquerading implementation deals with such protocol-specific features in separately loadable modules. Another problem is that masquerading should operate on transport level connections, whereas it is implemented in the network layer. The current implementation tries to address this with a limited session administration, but there are still some weaknesses to work on.

Masquerading can be enabled by specifying a special option with a forward filter rule. The next command creates a rule that makes every outgoing

telnet session being masqueraded (given that our local network has address 192.168.37.0):

```
ipfwadm -F -a accept -m -P tcp -S 192.168.37.0/24 \
        1024:65535 -D any/0 telnet
```

The `-m` flag in combination with the `accept` policy means: the packet is accepted (that is, allowed to be forwarded), but it gets masqueraded before being sent out. Because the masquerading mechanism depends on port numbers, it only works for TCP or UDP packets. So, be careful when using commands like:

```
ipfwadm -F -a accept -m -S 192.168.37.0/24
```

This command creates a rule that will cause all outgoing TCP and UDP traffic to be masqueraded. But it will also let all other packets (like ICMP messages) be forwarded unchanged, because they will also match with this rule! So, it's probably better to explicitly handle those cases, like with:

```
ipfwadm -F -p deny
ipfwadm -F -a accept -m -P tcp -S 192.168.37.0/24
ipfwadm -F -a accept -m -P udp -S 192.168.37.0/24
```

Especially when using unregistered IP addresses on your internal network (like the addresses defined in *RFC1918* or, even worse, illegally used addresses), no packets should ever be forwarded directly.

Please note that there are no "masquerading rules", but only forwarding rules with the masquerade flag set. So, you can list the rules with a command like:

```
ipfwadm -F -l
```

which will (given the above example) result in something like:

```
IP firewall forward rules, default policy: deny
type  prot source             destination        ports
acc/m tcp  192.168.37.0/24    anywhere           any -> any
acc/m udp  192.168.37.0/24    anywhere           any -> any
```

Besides this static information, the list of sessions currently being masqueraded can be inspected. This is dynamic information, changing every moment, which can be used to keep track of the external connections being active. The command

```
ipfwadm -M -l
```

might for example produce the following output:

```
IP masquerading entries
prot expire   source          destination      ports
tcp  13:00.15 int1.foo.com    ext2.bar.com     1017 (60001) -> login
tcp  14:15.60 int2.foo.com    ext1.bar.com     1346 (60010) -> telnet
tcp  14:52.82 int1.foo.com    ext1.bar.com     1348 (60015) -> ftp
```

The above table shows three sessions being masqueraded. The information is read from the pseudo-file `/proc/net/ip_masquerade`, which is converted to a human-readable format by *ipfwadm*.

7 IP traffic accounting

In Linux, IP traffic can be counted using accounting rules, defined by the same characteristics as the firewall rules. Accounting is done at two places: when a packet is received and when a packet is sent out (see figure 3).

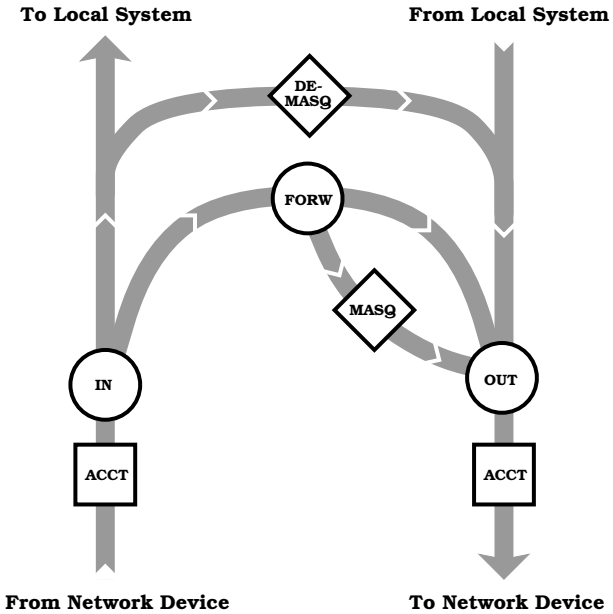

Figure 3

So, a packet being forwarded is counted twice: the first time just after its arrival, the second time when it is being sent out again. There is one single list of accounting rules, that is being used for both incoming and outgoing

traffic. For every packet, all rules in this list are checked and the packet and byte counters of every matching rule are incremented. Note the difference with the firewall lists: scanning a list there stops at the first match.

The following *ipfwadm* command counts all *http* traffic related to people using your WWW-server from the outside:

```
ipfwadm -A -a -b -W eth1 -P tcp -D 192.168.37.1 www
```

Here it is assumed that the local system, hosting the WWW-server, has IP address 192.168.37.1. We see some new options in this command. The `-b` option means "bidirectional", and makes that also packets *coming from* 192.168.37.1 (port 80) are counted. In general, think of the same rule with `-S` and `-D` swapped. The `-W` option has an interface name as parameter, so that only traffic via that particular interface is taken into account. Packets passing another interface (e.g., an interface *eth0* connected to your internal network) are not counted here.

Although there is only one list of accounting rules, used in both directions, it is possible to let a rule match only with incoming or outgoing packets:

```
ipfwadm -A in -a -W eth1 -P tcp -D 192.168.37.1 www
ipfwadm -A out -a -W eth1 -P tcp -S 192.168.37.1 www
```

The desired direction, `in`, `out`, or `both`, can be specified after the `-A` option. The default direction is `both`.

Some suggestions to use accounting most effectively:

- Reset the counters at certain intervals via *cron*, for example every day or every hour. Currently, the counters are 32-bits unsigned integers, so they will wrap around after 4 GByte.
- Combining the `-l` (list) and `-z` (reset to 0) options will reset the counters immediately after showing their current values, so no packets will be left uncounted.
- The most extensive output you get with the `-Alex` set of options, optionally with `-z` (reset values) or `-n` (numeric output) added.
- If you want to count all traffic excluding some specific protocols, like *http* and *ftp*, you can define rules for counting all traffic and rules for counting those protocols. Then just subtract the values from each other.

When listing the accounting rules (and the associated values) with *ipfwadm*, the pseudo-file `/proc/net/ip_acct` is read.

8 A real-life example

This section lists a complete example for a set of firewall filters on a Linux system acting as a gateway between the Internet and a private network. Note that this example is only included for illustrative purposes. Although it will protect the internal network to some extend, we strongly discourage to consider this to be a complete, robust firewall solution.

The example applies to a gateway system (*gw.foo.com*) connected to the Internet using interface *192.168.22.15* and to an internal network (*192.168.37.0*) via interface *192.168.37.1*. The system is a public WWW and ftp server, it can send and receive mail, it acts as a mail relay host for the internal network, and it is the primary DNS server for the *foo.com* domain.

Hosts on the private network can directly use telnet, WWW, ftp, gopher and WAIS services on the Internet (which is *not* a recommended firewall architecture). Also, ICMP traffic is allowed without any restrictions (e.g., to enable *ping*). Note that *traceroute* will not work, because this is using UDP packets to some unpriviliged ports.

```
# Some definitions for easy maintenance.
LOCALHOST="gw.foo.com"
LOCALNET="192.168.37.0/24"
IFEXTERN="192.168.22.15"
IFINTERN="192.168.37.1"
ANYWHERE="any/0"
UNPRIVPORTS="1024:65535"

# ====== Basic rules.

# Sure we're paranoid, but are we paranoid enough?
ipfwadm -I -p deny
ipfwadm -O -p deny
ipfwadm -F -p deny

# Refuse spoofed packets.
ipfwadm -I -a deny -V $IFEXTERN -S $LOCALNET
ipfwadm -I -a deny -V $IFEXTERN -S $IFEXTERN

# Unlimited traffic within the local network.
ipfwadm -I -a accept -V $IFINTERN
ipfwadm -O -a accept -V $IFINTERN

# Unlimited ICMP traffic (not recommended).
ipfwadm -I -a accept -P icmp
```

```
ipfwadm -O -a accept -P icmp
ipfwadm -F -a accept -P icmp

# ====== External use of our system.

# Public access for e-mail, ftp, WWW, and DNS.
ipfwadm -I -a accept -P tcp \
        -D $LOCALHOST smtp ftp www domain
ipfwadm -I -a accept -P udp -D $LOCALHOST domain
ipfwadm -I -a accept -k -P tcp \
        -D $LOCALHOST ftp-data
ipfwadm -O -a accept -P tcp -S $LOCALHOST smtp ftp \
           ftp-data www domain
ipfwadm -O -a accept -P udp -S $LOCALHOST domain

# ====== Internal use of the Internet.

# Outgoing packets.
ipfwadm -O -a accept -P tcp -S $LOCALNET $UNPRIVPORTS \
        -D $ANYWHERE smtp ftp ftp-data www telnet gopher \
           z3950 domain
ipfwadm -O -a accept -P tcp -S $IFEXTERN $UNPRIVPORTS \
        -D $ANYWHERE smtp ftp ftp-data www telnet gopher \
           z3950 domain
ipfwadm -O -a accept -P udp -S $LOCALNET $UNPRIVPORTS \
        -D $ANYWHERE z3950
ipfwadm -O -a accept -P udp -S $LOCALHOST $UNPRIVPORTS \
        -D $ANYWHERE z3950 domain
ipfwadm -F -a accept -P tcp -S $LOCALNET $UNPRIVPORTS \
        -D $ANYWHERE ftp ftp-data www telnet gopher z3950
ipfwadm -F -a accept -P udp -S $LOCALNET $UNPRIVPORTS \
        -D $ANYWHERE z3950

# Incoming packets.
ipfwadm -I -a accept -k -P tcp \
        -S $ANYWHERE ftp www telnet gopher z3950 domain \
        -D $LOCALNET $UNPRIVPORTS
ipfwadm -I -a accept -k -P tcp \
        -S $ANYWHERE ftp www telnet gopher z3950 domain \
        -D $IFEXTERN $UNPRIVPORTS
ipfwadm -I -a accept -P tcp \
        -S $ANYWHERE ftp-data -D $LOCALNET $UNPRIVPORTS
ipfwadm -I -a accept -P tcp \
        -S $ANYWHERE ftp-data -D $IFEXTERN $UNPRIVPORTS
```

```
ipfwadm -I -a accept -P udp \
        -S $ANYWHERE z3950 -D $LOCALNET $UNPRIVPORTS
ipfwadm -I -a accept -P udp -S $ANYWHERE z3950 domain \
        -D $LOCALHOST $UNPRIVPORTS
ipfwadm -F -a accept -k -P tcp \
        -S $ANYWHERE ftp www telnet gopher z3950 \
        -D $LOCALNET $UNPRIVPORTS
ipfwadm -F -a accept -P tcp \
        -S $ANYWHERE ftp-data -D $LOCALNET $UNPRIVPORTS
ipfwadm -F -a accept -P udp \
        -S $ANYWHERE z3950 -D $LOCALNET $UNPRIVPORTS
```

Some further remarks about the above example:

- The variable `$LOCALHOST` should only be used if this name will map to the two corresponding IP addresses, `$IFINTERN` and `$IFEXTERN`, via the */etc/hosts* file, without consulting a nameserver. The *ipfwadm* command will automatically create the necessary number of filter rules when a given hostname results in more than one IP address.
- A few rules are needed for all hosts on the internal network and for the gateway system itself. In these cases, two rules are created, one with `$LOCALNET` and one with `$IFEXTERN`, together covering all needed addresses.
- Adding some additional services, like *rlogin*, requires allowing a wider range of ports than defined in `$UNPRIVPORTS`. For these kind of services, the clients use ports below 1024.

9 Future directions

Although the current Linux firewall facilities are very useful, there are still some weaknesses and missing features. Therefore, possible areas for improvement in future Linux versions might be:

- Dynamic packet filters, so that temporary filter rules will be added automatically for specific protocols (like ftp).
- Some network address translation (NAT) mechanism.
- Further modularization of the firewall and masquerading code.
- Keeping some kind of state information, at least to detect related fragments.

- One or more new policies might be added for refusing a packet.
- Some graphical user interface for configuring the firewall and accounting rules, being a front-end to *ipfwadm*.

Given this, there is a good chance that Linux will soon be able to compete with the more advanced commercial firewall solutions on the market.

10 Complete firewalls based on Linux

In general, most of the optimal firewall solutions are some mixture of IP filters and application-level proxies, although the detailed architecture highly depends on the target environment. Proxies are mostly used for enabling extra authentication methods and advanced logging facilities. Linux systems can be used as a complete firewall solution, when using additional packages like the *Firewall Toolkit* of Trusted Information Systems, Inc. (TIS), *S/Key* or one of its derivatives like *OPIE* (One-time Passwords In Everything), *SOCKS*, etc. These packages are freely available and are known to work on Linux. But keep in mind that the use of a single bastion host without a filtering router is not recommended practice.

The pro's en con's of using free software for mission-critical applications like a firewall often focus on the *black box* versus *crystal box* debate. A commercial product usually doesn't come with source code, so nobody can study that code to find security leaks and try to misuse them. This is the black box concept, sometimes called "security by obscurity". On the other hand, when using freely available software, everybody, including the user, can look for possible bugs or weaknesses. The source code is available, so this approach is often referred to as a crystal box (of course, configuration details, local enhancements, etc., are *not* known publically). This usually will make bugs be found earlier, and fixes are just made available via the Internet, often within hours after detecting a bug.

The costs of a Linux firewall solution are relatively low. Given today's decreasing prices of PC's, a single Linux PC acting as a filtering router might cost less than US $ 1,000. A more advanced Linux system, also hosting several proxy services, a WWW-server, etc., will cost no more than US $ 2,000. Of course, these prices do not include consulting services to configure a customized firewall solution. But, be aware of the fact that you'll also have additional costs when buying a commercial firewall product. The quality of a total firewall solution highly depends on a well-designed configuration scheme, no matter whether it's based on commercial or free components.

A References

The newest Linux kernel is available from:

`ftp://ftp.cs.helsinki.fi/pub/Software/Linux/Kernel/`

The home page of *ipfwadm* on the World Wide Web:

`http://www.xos.nl/linux/ipfwadm/`

The *ipfwadm* package can be obtained via:

`ftp://ftp.xos.nl/pub/linux/ipfwadm/`

Some recommended books about firewalls:

- Building Internet Firewalls
  D. Brent Chapman, Elizabeth D. Zwicky
  O'Reilly & Associates, Inc.
  ISBN 1-56592-124-0

- Firewalls and Internet Security
  William R. Cheswick, Steven M. Bellovin
  Addison-Wesley Publishing Company
  ISBN 0-201-63357-4