Protecting Linux systems against attacks: part 2

CLOSE BULKHEADS!

MIRKO DÖLLE

After the mayhem we caused in part one, where we got rid of most daemons, we will now build a simple firewall that should insulate us against the last remaining few gaps.

***Firewall**: firewalls are used wherever private networks meet public ones, for example on company servers providing Internet access. Firewalls are meant to ensure that unauthorised access to the internal, private area is impossible. Depending on the complexity and size of the network, set up can take several days. However, firewalls are also sensible for domestic use if you want to protect your own computer against attacks from the Internet.*

***Masquerading**: primarily used on servers providing Internet access for local networks. Masquerading assigns the server's IP address to all queries from internal networks. The replies are translated back, so that internally there is no apparent difference between masked and unmasked connections. However, the local machines are not accessible from outside, as their IP addresses are not revealed, and queries can therefore only be made to the masquerader's IP, which ends up on the server itself. Masquerading is commonly used for leased line or flat rate connections that are used by more than one machine. Providers normally only give out one IP address per connection, which can only be used to address one machine. All other machines use private IP addresses and the masquerader attaches their own IP to Internet queries and handles the delivery of replies.*

Assuming that you have switched off the most important (or most useless) services, as described in the last issue, we can now turn to protection for the remaining daemons. We will achieve this by means of a **firewall** that controls outside access.

The subject of firewalls is notorious for being extremely complex - unfortunately with some justification - but for home use you'll be up and away with only a few lines. Administrators configuring large servers for companies or providers need to take many peripheral conditions and special services into account that are of no, or only minor, importance to home users.

For our example, we are using a computer dialing to the Internet via an ISDN card. Our interface is *ippp0* and the IP address assigned by our Internet provider is 192.168.1.1. Modem users simply need to leave out the 'i' in the device name; the modem interface is normally called *ppp0*. Network cards can also be used in the same way, but substituting *eth0*. The procedure itself is always the same.

The firewall acts as a filter between network devices, such as modems or ISDN and network cards, and the internal area. Which data ends up where is determined using filter rules. There are four

basic areas for these rules: all rules entered in the *input* section are applied to any incoming data packets in sequence, like a chain, while the rules under *output* are applied in turn to any outgoing data packets. The *forward* rules are used particularly for **masquerading**. In the fourth area it is possible to set up your own sections and rules. This is not normally required for home use, and we will deal with masquerading in a separate article.

The standard kernel of most distributions already contains firewall support, so no new compilation is necessary. The required package *ipchains* is set up for virtually all standard installations; if not, it can be found among the network utilities and installed personally.

We want to close the bulkheads and only give access to a few selected services. There are fundamental disadvantages to this method, which we will discuss in detail when looking at the respective rules. You can examine the rules that have been set up at any point using *ipchains -n -L*. To start with, everything is permitted:

```
linux:~ # ipchains -L
Chain input (policy ACCEPT)
Chain forward (policy ACCEPT)
Chain output (policy ACCEPT)
```

Policy describes the basic attitude towards data packets. When all rules in the chain have been applied to the packet without it being re-directed somewhere else, it is accepted with *ACCEPT* or discarded with *DENY*. Our aim is to deny everything that is not expressly permitted – therefore we will set the input policy to *DENY*:

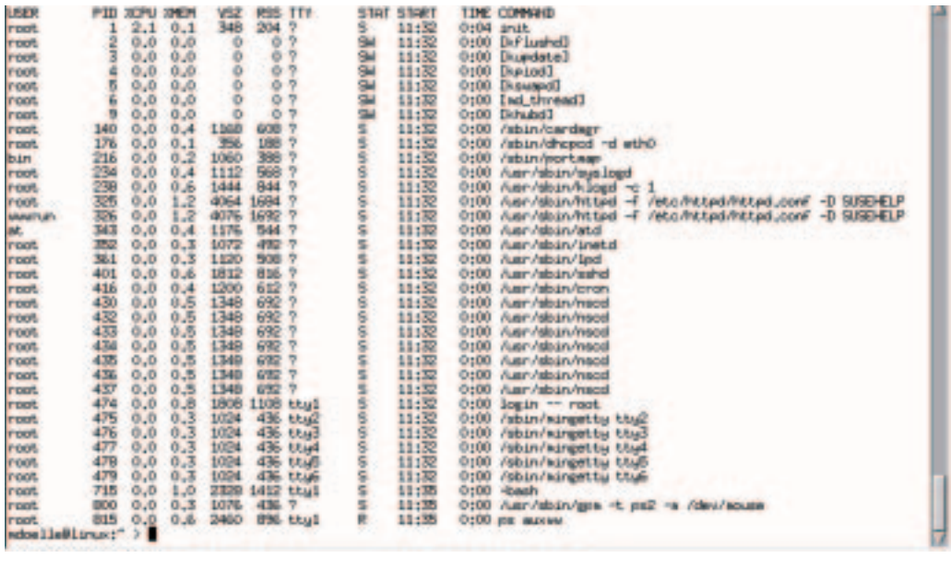
```
ipchains -P input DENY
```

Rules are always processed in sequence of entry, so we need to specify what we will accept from *ippp0* before discarding the rest. When a rule that obviously does not contain any errors doesn't work, it is usually due to an incorrect sequence. Once a packet has been discarded you cannot get it back in the next rule.

Now, nothing is working at all: any data is discarded, no matter whether received via the network, ISDN, modem or locally. In order to be able to use all our local services, and to keep our graphical interface working, we must except ourselves from being discarded:

```
ipchains -A input -i lo -j ACCEPT
```

***ICMP**: Internet Control Message Protocol – used in case of unavailability to send an appropriate message to the originator of a query. For instance, ping sends small data packets with ICMP echo-request (request for return) to the destination, in order to receive back the same data, via ICMP echo-reply (reply). This allows it to calculate the time lag between send and receive.*



[top]
Figure 1: Colourful activities: Most daemons can be recognised by the 'd' on the end, but also include portmap, cardmgr and cron

[below]
Figure 2: Utilities which (almost) no-one needs: In *letc/inetd.conf* too, there are hidden daemons, which are started completely automatically

The parameter *-A* specifies that we are adding a rule, *input* indicates the required section: all incoming data. Then follows the actual filter rule. *-i lo* applies to any data coming in via the loopback device, which can only be accessed by programs running locally on our machine and seeking a connection to other programs or services on our computer. Finally, with *-j* we stipulate what happens to the packets: they will be accepted.

Sealing ourselves off completely doesn't only have positive effects. For instance, we will no longer receive messages when we cannot reach a server. However, these messages are very important for smooth Internet traffic. Consequently we will permit them initially, from any direction:

```
ipchains -A input -p icmp -j ACCEPT
```

The parameter *-p icmp* indicates the **ICMP** protocol, responsible for transferring these messages; *-j ACCEPT* again represents the processing: accept.

Clear nameserver access

Another very important service is the *Domain Name Service*, or *DNS* for short. The DNS servers, nameservers for short, handle the resolution of, for example, *www.linux-magazine.co.uk* to the server's IP address, in this case *195.99.156.130*. Without this IP address you won't get anywhere on the Internet, so we must give access to our nameservers.

You will need the script in Listing 1, which you should save as *resolv-list* in the directory */usr/local/bin*. Please don't forget to make it executable with *chmod a+x /usr/local/bin/resolv-list*. *resolv-list* provides us with a list of nameservers used, which we then make accessible in our firewall using the following commands:

```
for ns in `/usr/local/bin/resolv-list`; do
  ipchains -A input -s $ns 53 -d 192.168.1.1 ➥
1024: -i ippp0 -p udp -j ACCEPT
  ipchains -A input -s $ns 53 -d 192.168.1.1 ➥
1024: -i ippp0 -p tcp -j ACCEPT
done
```

The difference between the two *ipchains* lines is in the protocols specified with *-p*, in this case **UDP** and **TCP**. The nameserver in our example is 192.168.2.1. Your IP will be different, depending on your Internet provider. You can enter several nameservers. Linux can cope with up to three.

The data source address is specified with *-s*. In our example the variable *$ns* was entered. After that follows the port number, or service ID, "53". Finally we name the destination, *-d*, with the possibility of restricting the permitted range of port numbers. By entering *1024:* we are permitting any port number from 1024 upwards to 65535. Ports below 1024 have a special status, but more about that later.

If you now enter *ipchains -n -L*, you should see the list in Listing 2 on the left-hand side of the page. Don't be put off by the second line. Even though it looks like everything is permitted everywhere, this is not the case. This output format does not display the device name to which the rule refers, and during set up we had specified the local loopback device with *-i lo*.

Listing 1: */usr/local/bin/resolv-list*

```
if [ -r /etc/resolv.conf ]; then
  set – `grep -i nameserver /etc/resolv.conf`
  while
    [ $# -ge 2 ];
  do
    echo $2
    shift 2
  done
fi
```

Listing 2:

```
Chain input (policy ACCEPT)
target  prot    opt    source  destination    ports
ACCEPT  all     ——  0.0.0.0/0      0.0.0.0/0      n/a
ACCEPT  icmp    ——  0.0.0.0/0      0.0.0.0/0      * -> *
ACCEPT  udp     ——  192.168.2.1    192.168.1.1    53 -> 1024:65535
ACCEPT  tcp     ——  192.168.2.1    192.168.1.1    53 -> 1024:65535
Chain forward (policy ACCEPT)
Chain output (policy ACCEPT)
```

***UDP**: User Datagram Protocol – a connectionless protocol, which means that data packets are not acknowledged by the recipient. The sender also doesn't repeat the data. This is used, for example, when querying DNS servers to find out the IP address associated with a host name. UDP is very fast, as no connection is established. UDP data cannot be sent directly through the Internet and are therefore normally wrapped in IP packets.*

***TCP**: Transfer Control Protocol – a frequently used Internet protocol. It is often wrongly referred to as TCP/IP, even though these are two protocols (TCP and IP). TCP ensures, among other things, that data is assembled in the correct order.*

***IP**: Internet Protocol – ensures the transfer of data packets on the Internet. This is where the IP addresses come in, which uniquely identify sender and recipient. UDP, ICMP and TCP data packets are wrapped in IP packets and provided with the addresses of sender and recipient before being sent through the Internet.*

Access encouraged

We also want to permit known users to log onto our system. In order to stop user names and passwords from being captured we will only allow the use of the encrypted *Secure Shell* or *SSH* for short. We deliberately spared the relevant daemon, *sshd*, when we were killing daemons in the last issue. Access is given using the rule:

```
ipchains -A input -d 192.168.1.1 ssh -p tcp -➥
i ippp0 -j ACCEPT
```

The parameter *-i ippp0* makes the rule applicable to any data coming in via the ISDN card. If we had another network card with further Linux machines attached to it, no one could log on to the system from those, as the rule is restricted to the ISDN card and we are, by default, rejecting anything else.

This rule will admit any data packets destined for the SSH service of machine 192.168.1.1 and entering the system via the ISDN card *ippp0*. As the packet has now been accepted, no other rules will be applied. *ipchains -n -L* now gives us:

```
Chain input (policy ACCEPT)
target  prot    opt     source  destination   ➥
ports
ACCEPT  all     ——  0.0.0.0/0       0.0.0.0/0➥
n/a
ACCEPT  icmp    ——  0.0.0.0/0       0.0.0.0/0➥
* -> *
ACCEPT  udp     ——  192.168.2.1     192.168.1➥
.1    53 -> 1024:65535
ACCEPT  tcp     ——  192.168.2.1     192.168.1➥
.1    53 -> 1024:65535
ACCEPT  tcp     ——  0.0.0.0/0       192.168.1➥
.1    * -> 22
Chain forward (policy ACCEPT)
Chain output (policy ACCEPT)
```

Web server access

If we want to make our Apache Web server accessible from outside, we require another ACCEPT rule:

```
ipchains -A input -d 192.168.1.1 http -p tcp ➥
-i ippp0 -j ACCEPT
```

As you can see, the pattern is the same, only the service entry has changed. The rule listing is extended by one line:

```
Chain input (policy ACCEPT)
target  prot    opt     source  destination   ➥
ports
ACCEPT  tcp     ——  0.0.0.0/0       192.168.1➥
.1    * -> 80
```

Ports and services

Anything that is not permitted is denied. This, at the moment, includes anything that is not a nameserver reply or SSH connection – even standard surfing activities. So we will have to consider what else we need to permit, to enable normal operations. This is not possible without knowledge of ports.

Behind the entries for services such as *ssh* or *http* in our examples lie the port numbers. In the example of how to give nameserver access we actually worked directly with the port number, 53.

Imagine a large block of flats in which all the letter boxes have been numbered sequentially – they all have the same address (IP), and letters can only be delivered correctly on the basis of the letterbox number (port number) or the name on the letterbox (service description). You can find out which service corresponds to which port number from the file */etc/services*.

Ports 0 to 1023 have a special role: these numbers are reserved for privileged services. The daemons behind them are normally running with root privileges. These ports are generally not available to normal users.

Replies to Netscape queries always originate from ports starting at 1024. We still need to give access to these. However, we can restrict the whole thing a bit further: it is not necessary during surfing for anyone to connect to us, as we are querying the server and it returns the reply through the same connection. Incoming connection requests are therefore not accepted (*! -y*):

```
ipchains -A input -d 192.168.1.1 1024: -i ipp➥
p0 -p tcp -j ACCEPT ! -y
```

There is still one catch: the user's SSH client will normally try to open a second channel in the range of ports 600 to 1023 once it has logged onto the server. This is no longer possible, as everything up to port 1023 has been sealed off. For some helpful advice, see the SSH and Firewall box.

Practical effects

To summarise: we are accepting SSH connections through the ISDN card, as well as requests to our Apache Web server. ICMP messages, DNS server replies and requested Internet data are also let through. On the other hand, any external connection that is not routed through SSH or the Web server will always be ignored.

These settings will only have a minor impact on the user sitting at their machine. Even if the *talk* daemon has not been switched off (as discussed in the last issue) users can no longer be addressed from the Internet. External administration via *swat* or *linuxconf* is not possible, however it is no problem from the user's own machine. The only limitation is with IRC: we can no longer send data

Table 1: Service access rules

Nameserver:
```
ipchains -A input -d IP 53 -p udp -i Interface -j ACCEPT
ipchains -A input -d IP 53 -p tcp -i Interface -j ACCEPT
```

SSH access:
```
ipchains -A input -d IP ssh -p tcp -i Interface -j ACCEPT
```

Telnet access:
```
ipchains -A input -d IP telnet -p tcp -i Interface -j ACCEPT
```

Sendmail access:
```
ipchains -A input -d IP smtp -p tcp -i Interface -j ACCEPT
```

Apache Web server:
```
ipchains -A input -d IP http -p tcp -i Interface -j ACCEPT
```

FTP access:
```
ipchains -A input -d IP ftp -p tcp -i Interface -j ACCEPT
ipchains -A input -s 0/0 ftp-data -d IP 1024: -p tcp -i Interface -j
ACCEPT
```

ICQ:
```
ipchains -A input -d IP 4000 -p tcp -i Interface -j ACCEPT
```

IRC with DCC:
```
ipchains -A input -d IP 1024: -p tcp -i Interface -j ACCEPT
```

This rule is to be used with care, as it allows an external connection to be established on non-privileged ports. If this rule is implemented, no other rule for TCP protocol and ports from 1024 upwards must be active.

SSH and firewall

SSH will normally try to establish a second channel through a port between 600 and 1023. However, as we have prevented this with the firewall set up in the article, SSH would not be able to connect. There are two solutions: either call SSH with the parameter '-P', or amend the rights for SSH. Normally any SSH connection is established with root permissions in order to be able to use a port below 1024. Using the command chmod u-s `which ssh', you can ensure that SSH will be started with your user rights in future – and automatically uses a port upwards of 1023 as the return channel.

Figure 3: Almost all utilities were superfluous: we need *http-rman* for the SuSE help system, *swat* stands in for the system administration program *linuxconf* of other distributions.

Info

Firewall manual by Guido Stepken with many examples: http://www2.little-idiot.de/firewall/

Notes and extensions by Dirk Haase for users of EasyLinux for the first part of Close bulkheads!: http://members.tripod.de/krids oft/easyll/ha/ha005.html

Figure 4: Activation and deactivation is done differently from one distribution to another – here for example in linuxconf under Red Hat (left) and DrakConf under Mandrake.

via DCC or otherwise. Our FTP server is also no longer accessible to outsiders.

Table 1 is a list of permission rules you can build into a firewall to allow access for individual services.

Automatic activation

A huge problem in building a domestic firewall is that your own IP address changes each time you log on - and consequently needs to be corrected in the firewall rules. Most firewall configuration tools make no provisions for changes in the IP address and are therefore not suitable for home use.

Ideally, the rules would be activated automatically after each login, with the correct IP, of course, and deactivated once you log off.

The required scripts into which we can integrate our rules are called */etc/ppp/ip-up* and */etc/ppp/ip-down*. *ip-up* is called as soon as login has occurred, and *ip-down* once you have logged off. We are making use of the fact that parameter $1 gives us the modem or ISDN interface and $4 our assigned IP address. Since the lines for set up and removal of the firewall rules are almost identical, we will combine the rules in the file */etc/ppp/inet_chains*, using the appropriate variables for IP and interface used. You can see a relevant example on the CD under *LinuxUser/firewall/inet_chains*. There you will also find the access rules mentioned in Table 1, commented out with a hash (#) at the beginning of the line and therefore not active. Should you want to give access to individual services you only need to remove the hash. The file */usr/local/bin/resolv-list* from Listing 1 is no longer needed for this by the way, *inet_chains* has its own function for this purpose.

The call to *inet_chains* should be entered near the start, preferably in the second line, of */etc/ppp/ip-up* and */etc/ppp/ip-down*:
In */etc/ppp/ip-up*,

```
test -x /etc/ppp/inet_chains && /etc/ppp/inet_chains up $@
```

In */etc/ppp/ip-down*,

```
test -x /etc/ppp/inet_chains && /etc/ppp/inet_chains down $@
```

Conclusion:

In regard to standard installations, distributors have a lot of catching up to do. Only Mandrake possesses a useful mechanism that will switch off virtually any service at a paranoid setting. With most other distributions even security profiles are little help. Distributions especially aimed at beginners, starting with the SuSE 7.0 personal edition, ought to be better suited to their end users' requirements. It must be hoped that the next versions from the big distributors will take this on board.

Nevertheless, no computer is really secure. Even if the possibilities described above provide you with reasonable external protection, one day the error that will invalidate everything may be found. And there is one thing you ought to know: the Internet is evil, and it gets everybody eventually. ■