Packet Filtering Firewall

INTRODUCTION

Pre-requisites

TCP/IP

NAT & IP Masquerade

Packet Filters vs Proxy Servers

Firewalls make a simple decision: accept or deny communication. There are two distinct types of firewalls: packet filters and proxy servers. The difference between the two types of firewalls lies in what information the firewall uses to make the accept/deny decision. The packet filter is the simpler of the two firewalls. The packet filter makes its decision using network information. By network information, I mean the information contained in the TCP, UDP, IP, and other protocol headers. The packet filter does not examine the data section of a packet. A proxy server, on the other hand, operates at the application level. For instance, an http proxy server firewall can make a decision to accept or deny communications based on the content of a web page. Packet filters are cheap, fast and easy to maintain. Proxy servers can make more informed decisions, but they are expensive, slower and much more difficult to maintain. This lesson explains how to implement a simple, packet filtering firewall.

Hardware

Packet filters require few hardware resources. Linux packet filtering firewalls have been successfully run on Intel 80386 machines with 8 MB RAM. However, some of the more complex filtering decisions require a math coprocessor and additional memory. Therefore I would recommend at least a Pentium 66 Mhz with 16 MB RAM with linux kernel 2.4.18 and later. No hard drive is required – the packet filter can boot from floppy or CD-ROM and mount its root directory as a ramdisk (see the floppy firewall lesson for more information).

Jargon

The computer industry usies a lot of jargon. Excessive and improper use of jargon presents a major obstacle to people who want to learn about computer technology. To make matters worse, industry people will use the same word to mean different things in different applications, or the different words meaning the same thing. Where ever possible I will try to unravel the jargon and explain what things mean. But if I use a term that you don't understand please let me know.

Versions

This lesson was developed using the following software:

- Linux kernel 2.4.18
- iptables 1.2.5

IP Forwarding

The function of a firewall is to take packets from one network, examine them, and retransmit them on another network. In linux kernel jargon, retransmitting packets is called “IP forwarding” (the terms “forwarding” and “routing” are often mixed up – read the endnote for an explanation). IP forwarding is built into the linux kernel, but it is turned off. To turn IP forwarding on and off, use the following commands:

```
echo “1” > /proc/sys/net/ipv4/ip_forward
echo “0” > /proc/sys/net/ipv4/ip_forward
```

PACKET FILTERING, NETFILTER AND IPTABLES

In the linux 2.4 kernel, packet filtering is executed by the netfilter module.The basic idea behind netfilter is that incoming and outgoing packets are tested by user-specified rules which determine what will happen to the packet.

Iptables

The security administrator uses the **`iptables`** utility to set the rules. When we talk about configuring the firewall, what we are really talking about is designing a set of rules then executing them using a series of **`iptables`** commands.. The command must be executed as root. Iptables takes a large number of options. Options are identified by a single or double “-” character. These options specify what you want to do: add a rule, delete a rule, add a chain, list rules, etc. For instance, to list all the rules use the command:

```
[root@Radagast andrew]# /sbin/iptables –list
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

In this case the rule table is empty because we haven't created any rules yet.

RULES

Rules describe the security policy of the firewall. More specifically, a rule tells the netfilter engine what to do with a packet. Each rule has two parts: (1) a description of a packet and (2) an action. If the packet matches the description, then the engine performs the action. We can also implement inverse matching, meaning that if the packet doesn't match the description, then do the action. In linux jargon, the description is called the criteria and the action is called the target.

Criteria

In the TCP/IP lesson we discussed the information that accompanies a packet. For instance, TCP segments and UDP datagrams include information about source and destination port. TCP segments contain connection information in their SYN and ACK bits, and in their SYN number and ACK number fields. IP datagrams contain source and destination address, fragmentation information, type of service and protocol. Ethernet frames carry source and destination MAC address. In addition to this information, the packet filtering software "knows" which network interface the packet was received from (the "in" interface) and which network interface the packet would be transmitted on (the "out" interface). All of this information may be used as criteria. Table 1 lists many commonly used criteria, the iptables command switch used to specify each criterion, and a short description of the criterion.

Table 1: Criteria (mostly copied from iptables man page)

In Interface	-i --in-interface	Name of an interface via which a packet is going to be received (only for packets entering the INPUT, FORWARD, and PREROUTING chains). If the interface name ends in a "+" then any interface which begins with this name will match. If this option is omitted, any interface name will match.
Out interface	-o --out-interface	Name of an interface via which a packet is going to be sent (for packets entering the FORWARD, OUTPUT and POSTROUTING chains). If the interface name ends in a "+" then any interface which begins with this name will match. If this option is omitted any interface will match.
Protocol	-p --protocol	The protocol of the rule or of the packet to check. The specified protocol can be one of *tcp*, *udp*, *icmp*, or *all*, or it can be a numeric value, representing one of these protocols or a different one. A protocol name from **`/etc/protocols`** is also allowed. The number zero is equivalent to *all*. Protocol *all* will match with all protocols and is taken as default when this option is omitted.
Source Address	-s --source	Source specification. Address can be either a network name, a hostname (please note that specifying any name to be resolved with a remote query such as DNS is a really bad idea), a network IP address (with /mask), or a plain IP address. The mask can be either a network mask or a plain number, specifying the number of 1's at the left side of the network mask. Thus, a mask of 24 is equivalent to 255.255.255.0.
Destination address	-d --destination	Destination specification. Similar to source address (see previous entry).
Fragment	-f --fragment	This means that the rule only refers to second and further fragments of fragmented packets. Since there is no way to tell the source or destination ports of such a packet (or ICMP type), such a packet will not match any rules which specify them. When the "!" argument precedes the "-f" flag, the rule will only match head fragments, or unfragmented packets.
Source Port	--sport --source-port	Source port or port range specification. This can either be a service name or a port number. An inclusive range can also be specified, using the format *port*:*port*. If the first port is omitted, "0" is assumed; if the last is omitted, "65535" is assumed. If the second port is greater than the first they will be swapped.
Destination Port	--dport --destination-port	Destination port or port range specification.
Syn	--syn	Only match TCP packets with the SYN bit set and the ACK and FIN bits cleared. Such packets are used to request TCP connection initiation; for example, blocking such packets coming in an interface will prevent incoming TCP connections, but outgoing TCP connections will be unaffected. It is equivalent to –tcp-flags SYN,RST,ACK SYN.
ICMP type	--icmp-type	This allows specification of the ICMP type, which can be a numeric ICMP type, or one of the ICMP type names shown by the command iptables -p icmp -h
MAC Address	--mac-source	Match source MAC address. It must be of the form XX:XX:XX:XX:XX:XX. Note that this only makes sense for packets coming from an Ethernet device and entering the PREROUTING, FORWARD or INPUT chains

Table 1: Criteria (mostly copied from iptables man page)

State	--state	Where state is a comma separated list of the connection states to match. Possible states are INVALID meaning that the packet is associated with no known connection, ESTABLISHED meaning that the packet is associated with a connection which has seen packets in both directions, NEW meaning that the packet has started a new connection, or otherwise associated with a connection which has not seen packets in both directions, and RELATED meaning that the packet is starting a new connection, but is associated with an existing connection, such as an FTP data transfer, or an ICMP error.
Limit	--limit *rate*	Maximum average matching rate: specified as a number, with an optional `/second', `/minute', `/hour', or `/day' suffix; the default is 3/hour.
	--limit-burst *number*	Maximum initial number of packets to match: this number gets recharged by one every time the limit specified above is not reached, up to this number; the default is 5.

Inversion

In addition to the criteria in Table 1, each criteria may be inverted using the "!" operator. For instance, "-s 192.168.1.0/24 -j DROP" means "drop packets coming from network 192.168.1", but " -s !192.168.0.24 -j DROP" means "drop packets that don't come from network 192.168.1".

Match Extensions

The netfilter engine was originally designed to filter using only information from the IP header and the in/out interface. But matching other information can be extremely useful. Therefore the criteria that netfilter can match has been extended – the extra criteria are called "match extensions." Match extensions are implemented as separate kernel modules. The modules may be loaded in two ways: implicitly when the "-p" or "--protocol" flag is specified, or explicitly using "-m *modulename*". For instance, most firewalls will use rules that match TCP segment header fields, especially destination port. To allow http, for instance, a security administrator might implement the following rule:

```
iptables -p TCP -dport 80 -j ACCEPT
```

The "-p TCP" option loads the TCP module, which executes the destination port match.

MAC Address

Every ethernet card has a Media Access Control (MAC) address. In theory, every MAC address is unique, however some ethernet cards allow user programmable MAC addresses. If your system includes an ethernet card, then you may view the MAC address using the ifconfig eth0 command:

```
[root@Radagast andrew]# /sbin/ifconfig eth0
eth0 Link encap:Ethernet  HWaddr 00:50:BA:70:0E:BA
     inet addr:192.168.0.1  Bcast:192.168.0.25 Mask:255.255.255.0
     UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
     RX packets:1236933 errors:0 dropped:0 overruns:0 frame:0
     TX packets:731462 errors:0 dropped:0 overruns:0 carrier:0
     collisions:22 txqueuelen:100
     RX bytes:1664437704 (1587.3 Mb)  TX bytes:71297062 (67.9 Mb)
     Interrupt:12 Base address:0xe400
```

The HWaddr field indicates the MAC address, in this case 00:50:BA:70:0E:BA. The netfilter can filter on MAC address, which can be a potent tool to lock down communication between specific computers on a broadcast network.

Connection State

As we discussed in the TCP/IP lesson, most internet applications use TCP which is a connection based protocol. Two computers using TCP establish a connection using SYN and ACK segments. Once the connection is established, the computers may exchange data reliably.

There are many instances where you might want to filter on TCP connection state information. Suppose that you want computers on your private network to be able to read web pages from the internet, but you don't want computers on the internet to read web pages located on your private network. The World Wide Web uses TCP port 80 to transfer information. So essentially, you want to be able to create TCP port 80 connections from your private network to the internet, but you want to deny TCP port 80 connections from the internet to computers on your private network. This requires the TCP connection state module, and the rule might look like this:

```
iptables -A FORWARD -m state –state NEW -i $PRIVATE -j ACCEPT
iptables -A FORWARD -m state –state NEW -i $INTERNET -j DUMP
```

In this example, we have used local variables to replace the interface names. Early in the firewall script there would have been a pair of definitions such as:

```
PRIVATE = eth0
INTERNET = ppp0
```

Variables like this are very common in firewall scripts. But back to the example. Our private network computers can send TCP segments which create new connections, but after the TCP modules establish the connection we expect to exchange data. So we need a rule accepting packets for established connections.

```
iptables -A FORWARD -m state –state ESTABLISHED,RELATED -j ACCEPT
```

If you are a good security administrator then you noticed that I snuck the RELATED match into the preceding rule – here's why: some TCP applications use two connections. For instance, when you connect to another computer using the file transfer protocol (ftp, RFC 959) you use TCP port 21 to connect from a computer on the private network to a ftp server on the internet. When the private network user request a file, the ftp server creates second connection from itself to the computer on the private network. The second connection is used to transfer the requested file. The problem is that the firewall does not allow new connections from the internet, so the server will not be able to create the second connection. The RELATED criteria matches these secondary connections that are related to existing connections. We will allow related connections.

Sometimes internet bad guys send bad TCP segments. Sometimes they do this as a sneaky port scan. Sometimes they do this to try to hijack an existing TCP connection. These invalid TCP segments are not trying to establish new connections, and they aren't part of an existing connection. We can use a state matching rule to drop invalid TCP connection attempts:

```
iptables -A FORWARD -m state –state INVALID
```

Limits

The limit match extension allows a match against the rate at which the firewall receives packets. For instance, the rule

```
iptables -A INPUT -m limit –limit 1/second \
                 -p TCP -dport 80-j ACCEPT
```

accepts one http packet per second. The limit module also understands “/minute”, “/hour”, and “/day”. We can also specify a burst-limit match, for instance

```
iptables -A INPUT -m limit –limit 1/second –limit-burst 5 \
                  -p TCP -dport 80 -j ACCEPT
```

accepts up to five http packets in the first second, then one packet per second after that. The burst recharges by one for each time unitin which the limit isn't met. For instance, if six packets arrived in the first second, the rule would match the first five packets but not the sixth. The burst value is now reduced to zero, because it was entirely used. If two packets arrived in the next second, the rule would only match one of them, because the burst has already been used and the average match is only one per second. If no packets arrived in the next second, then the burst value would recharge to one. If no packets arrived in the second after that, then the burst value would recharge again to two, and so on. The limit match can be used to protect a server from syn-floods:

```
iptables -A INPUT -p TCP –syn \
                  -m ! limit --limit 1/s --limit-burst 4 -j DROP
```

There are three criteria. The first criteria is that the packet contains a TCP SYN segment. The next two criteria are limits, but the limits are inverted. The inverted limit means that a match occurs if the packets exceed the limit. In other words, if we get a burst of more than 4 SYN segments in a second, or if we get an average of more than 1 SYN segment per second, then netfilter gets a match.

Target

Each rule specifies criteria and a target. The target is the action to perform if the match succeeds. Netfilter includes four built-in targets: ACCEPT, DROP, QUEUE, and RETURN (by convention, target names are written in capitals). In addition, the name of a custom-defined chain may be specified as a target (we will discuss chains later). Extension targets may be built into the kernel or loaded as modules – we will discuss two extension targets later in this lesson: LOG and MASQUERADE.

- ACCEPT means that the firewall approves the matching packet for transmission
- DROP means that the firewall has rejected the matching packet. The packet is “dropped on the floor”.
- QUEUE is not covered in this lesson.
- RETURN. We will discuss the RETURN target later in this lesson.
- LOG instructs netfilter to add an entry in the kernel log
- MASQUERADE. This target is only valid in the NAT table in the POSTROUTING chain. It should only be used with dynamically assigned IP connections: if you have a static IP address, you should use the SNAT target. Masquerading is equivalent to specifying a mapping to the IP address of the interface the packet is going out, but also has the effect that connections are forgotten when the interface goes down. This is the correct behaviour when the next dialup is unlikely to have the same interface address (and hence any established connections are lost anyway).

Example Rules

Here are some simple example rules. You should study each example carefully. Make sure that you understand what each argument means.

Suppose that you have a network, 204.101.3/24. You connect this network to the internet through a firewall. You don't want hackers on the internet pinging machines on your network. You know that ping is implement using the ICMP protocol, so you add the following rule:

```
iptables -A FORWARD -d 204.101.3.0/24 -p ICMP -j DROP
```

You are the sysadmin on the above network. Your firewall has lots of rules which specify what protocols are permitted, limits, etc, etc. But you want to create a back door so that your personal computer at 204.101.3.4 has unrestricted access to and from the internet. You put the following commands near the beginning of your firewall script:

```
iptables –A FORWARD –s 204.101.3.4 –j ACCEPT
iptables –A FORWARD –d 204.101.3.4 –j ACCEPT
```

You have implemented a wireless LAN at your office by adding a wireless ethernet card to your linux firewall/ router. But you don't want every kid in town accessing the internet through your system. You only have one wireless client, and you know its MAC addresses (a0:b1:c2:d3:e4:f5). So you add the following rule to your firewall:

```
iptables -A FORWARD -in-interface wlan0 -m mac \
                    -mac-source !a0:b1:c2:d3:e4:f5 -j DROP
```

You have a private network. Within this network you have some Microsoft Windows workstations. The Windows machines use Microsoft Networking. You decide that you don't want people on the internet to read shared files on your private network, so you use the following rule:

```
iptables -A FORWARD -p UDP -sport 137:139 -j DROP
```

CHAINS

Most firewalls will include many rules. The rules are organized in chains. A chain is simply an ordered sequence of rules. The iptables engine analyzes the packet using the first rule in the chain. If the packet matches the criteria of the first rule, then the engine performs the action described by the target; if the packet does not match the criteria, then the engine goes to the next rule in the chain (note that after performing some targets, e.g. LOG, netfilter will go to the next rule in the chain). If the packet “falls off the end of the chain” then netfilter behaves differently for user-defined and built-in chains. For built-in chains the behaviour is defined by the chain's policy.

Policy

The policy is a target defined by the iptables utility using the -P command. For instance, the command

```
iptables -P FORWARD DROP
```

means that packets which fall off the end of the FORWARD chain should be DROPPED. If the security administrator fails to specify a policy for the chain, then the filter engine uses the default policy for the chain which is always ACCEPT, which is very bad.

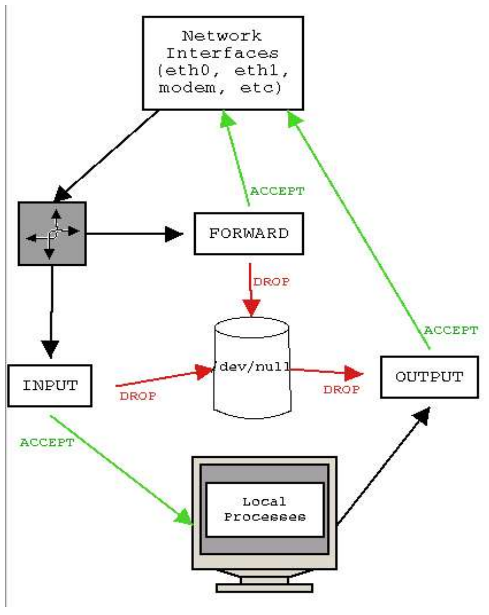

Figure 1: Filter Table

If a default policy of ACCEPT is very bad, then why did the creators of netfilter build it that way? Well, suppose that a new linux user accidentally turns on iptables. If the default policy were DROP, then every packet would be discarded. Many applications use sockets between different processes on the same machine, so even if the machine isn't connected to the internet, an accidental default policy of DROP with no rules to specify permitted communications would render many important applications useless. So the default policy is ACCEPT, and the first thing your firewall script should do is flush the chains and set the policy to DROP:

```
iptables -F FORWARD
iptables -F INPUT
iptables -F OUTPUT
iptables -P FORWARD DROP
iptables -P INPUT DROP
iptables -P OUTPUT DROP
```

The filter table has three built-in chains: INPUT, OUTPUT and FORWARD. Figure 1 depicts the built-in chains. In figure 1, the top of the diagram represents the network interfaces (ethernet cards, modems, serial interfaces, etc). This is where IP packets enter and exit the machine. Packets entering the machine first go to the forwarding process. It is important to remember the difference between routing and forwarding. The forwarding process compares the packet's destination IP address to the user-defined routing table; in this way the forwarding process assigns an output interface to the packet, or assigns the packet to a socket on the local host. But the routing process does not immediately retransmit the packet on that interface. Instead, the packet enters the netfilter. If the packet is destined for a local process then the netfilter uses the INPUT chain. For instance, the firewall machine might allow remote management using secure shell, and therefore might be running the secure shell daemon as a local process. But a firewall's job is to filter packets between different network interfaces so almost all packets are filtered using the FORWARD chain. If one of the rules ACCEPTs the packet, then the packet is sent to the network interfaces to be retransmitted. If one of the rules DROPs the packet (or if the packet falls off the end of the chain) then the packet goes to **`/dev/null`**.

User Defined Chains

In addition to the built-in chains, users may define their own. The sysadmin creates new chains using the iptables -N command. By convention, chain names always use capital letters.

```
iptables -N TCP_CHAIN
```

After creating the new chain, the user adds it as a target to one of the existing chains

```
iptables -A FORWARD -p TCP -j TCP_CHAIN
```

In the example above, we specify that if the packet contains part of a TCP segment, then netfilter should examine the packet using the rules in the TCP_CHAIN. So now we need to add some rules to the new chain.

```
iptables -A TCP_CHAIN -dport 22 -j RETURN
```

RETURN is a special target. In a user defined chain, RETURN tells netfilter to stop processing the packet on this chain and return to the original chain. If there is no original chain, i.e. if RETURN is used in a built-in chain, then netfilter applies the default policy to the packet. We will add some more rules to the new chain.

```
iptables -A TCP_CHAIN -dport 80 -j ACCEPT
iptables -A TCP_CHAIN -dport 21 -j ACCEPT
iptables -A TCP_CHAIN -dport 25 -j ACCEPT
```

User defined chains do not have policies. If a packet drops off the end of a user defined chain, then netfilter continues processing the packet using the original chain. But we can implement a drop policy by adding a rule with an empty criteria. An empty criteria matches everything, so the rule

```
iptables -A TCP_CHAIN -j DROP
```

will drop every packet. We add this rule to the end of the TCP_CHAIN. Our custom chains says that TCP segments to port 22 (secure shell) will be evaluated later in the original chain, segments to ports 21 (ftp), 25 (smtp), and 80 (http) will be accepted, and TCP segments to all other ports will be dropped.

Here is another example of a user defined chain. Remember the rule to protect against SYN floods:

```
iptables -A INPUT –p TCP –syn \
                  -m ! limit --limit 1/s --limit-burst 4 -j DROP
```

Now suppose that in addition to dropping the packets, we want to print a warning in the kernel log. First we create the chain:

```
iptables -N SYN-FLOOD
iptables -A SYN-FLOOD -m limit --limit 1/s --limit-burst 4 -j
RETURN
iptables -A SYN-FLOOD -j LOG –log-prefix “** SYN FLOOD **”
iptables -A SYN-FLOOD -j DROP
```

Then we add a rule to one of the built-in chains forcing netfilter to examine all SYN packets using this new chain:

```
iptables -A INPUT -p TCP –syn -j SYN-FLOOD
```

TABLES

Chains are grouped into tables. There are three tables: filter, NAT and mangle.

Filter Table

In the previous section, drawn in Figure 1, we talked about the INPUT, OUTPUT, and FORWARD chains. These chains belong to the filter table. Filter is the default table, so if you don't specify a table then iptables will assume that whatever you want to do (add a chain, append a rule to a chain, etc) is being done within the filter table. If you want to modify one of the other tables, then you must use the iptables utility with the -t table_name option.

NAT Table

Netfilter implements NAT and IP masquerade using the NAT table.

Mangle Table

Netfilter implements packet mangling using the mangle table. Packet mangling is an advanced technique whereby arbitrary TCP/UDP/IP headers may be modified. This lesson does not cover packet mangling.

Relationship Between Tables

There are three tables, and each table has at least three chains, so you might wonder which table gets evaluated first, and which chain gets evaluated first in each table. The answer is complicated: the chains from the tables are all mixed up, so a packet might first be evaluated against the PREROUTING chain from the MANGLE table, then against the PREROUTING chain of the NAT table, then the FORWARD chain of the FILTER table, then the POSTROUTING chain of the NAT table. But it's more complicated than that – netfilter evaluates the packet against different chains depending on where the packet comes from (a network interface or a local process) and where the packet is going (a network interface or a local process).

The flow chart in Figure 2 illustrates the relationship between the different tables and chains. Packets may enter the flow in two places: packets may arrive at one of the network interfaces at the top, or packets might be generated by a local process (left middle). A local process is an application that is running on the firewall, such as secure shell.

Once a packet has entered the flow chart, it proceeds in the direction of the arrow to the next box. Each box represents a chain of rules. I have colour coded the boxes: blue boxes belong to the mangle table, yellow boxes to the NAT table and green boxes belong to the filter table. On a firewall most packets will enter the flow from a network interface, so let's consider a packet as it enters the flowchart from the top. Netfilter receives the packet from the IP module and examines the packet using the rules in the PREROUTING chain of the mangle table, then the PREROUTING chain of the NAT table. Usually we don't make any changes to the NAT PREROUTING chain - this is where netfilter automatically looks up the packet's destination socket in the IP Masquerade and Port Forwarding tables. If there is a translation, then netfilter replaces the destination Internet IP socket with the private network socket. Then the packet goes to the IP forwarding module.

The IP routing module decides where the packet will go: to the ip forwarding module or to a local process. If the firewall is running local processes such as secure shell, then some packets may traverse the FILTER:INPUT, MANGLE:OUTPUT, NAT:OUTPUT, and FILTER: OUTPUT chains. But typically these OUTPUT chains are used on a workstation to provide extra security, not on a firewall. Firewalls specialize in forwarding packets, so the routing decision is almost always to forward the packet. Looking at the flow chart we see that the netfilter module will apply the rules in the FILTER:FORWARD chain, and finally the rules in the NAT:POSTROUTING chain. Most of the firewall's rules will be in the FILTER:FORWARD chain – this is where we specify the default policy of

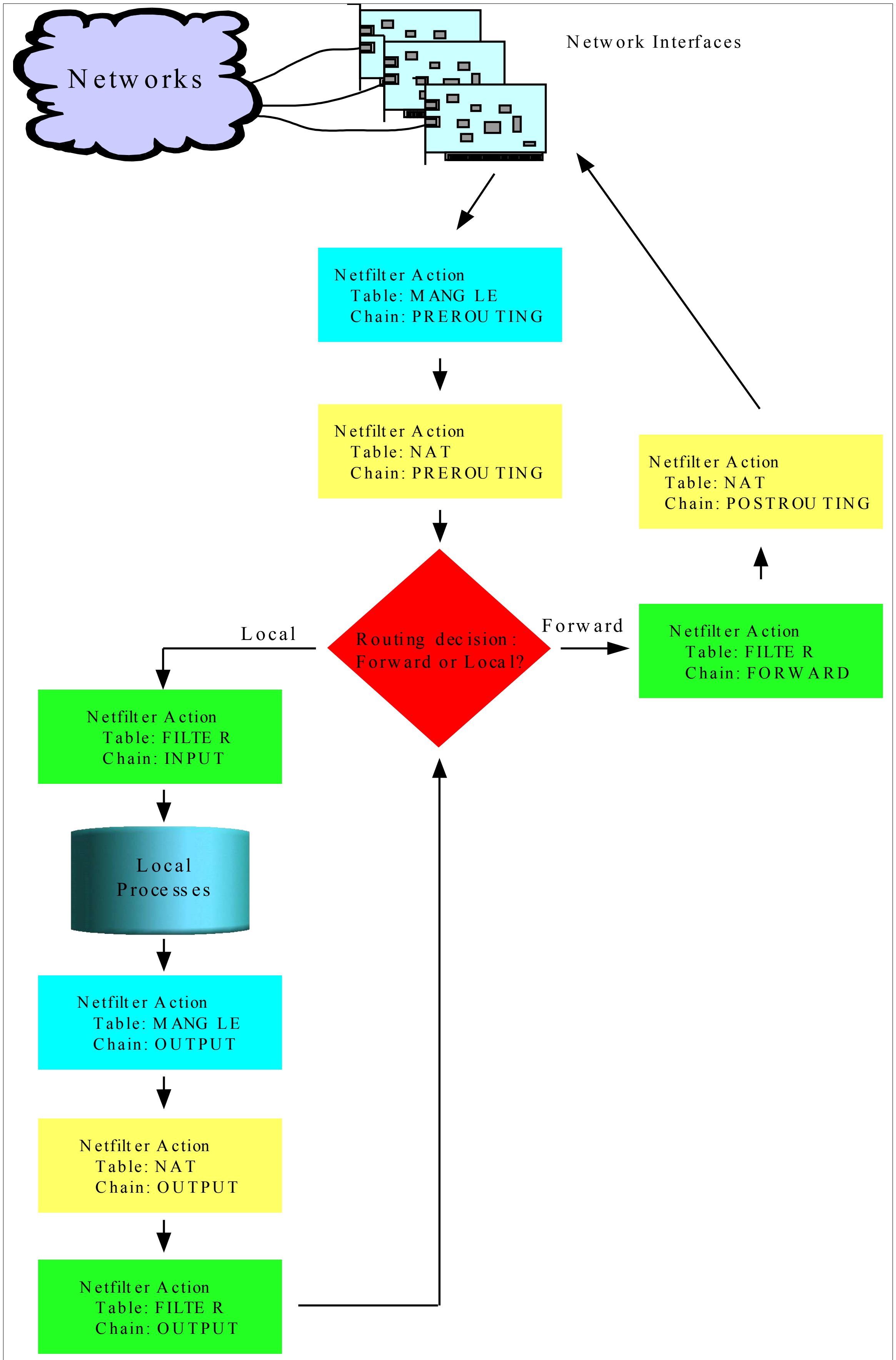
Networks
Network Interfaces
Netfilter Action Table: MANGLE Chain: PREROUTING
Netfilter Action Table: NAT Chain: PREROUTING
Routing decision: Forward or Local?
Local
Forward
Netfilter Action Table: FILTER Chain: FORWARD
Netfilter Action Table: NAT Chain: POSTROUTING
Netfilter Action Table: FILTER Chain: INPUT
Local Processes
Netfilter Action Table: MANGLE Chain: OUTPUT
Netfilter Action Table: NAT Chain: OUTPUT
Netfilter Action Table: FILTER Chain: OUTPUT

Figure 2: Relationship Between Tables

DROP and describe which packets we will ACCEPT. The NAT:POSTROUTING chain is important because that is where we implement IP masquerade, like this:

```
iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE
```

Suppose that eth1 is the interface to the Internet. The rule above specifies that netfilter will implement IP masquerade for any packet going to the Internet.

There is a general principal that your firewall should not run any internet services. So no local processes should generate or receive packets. And we will ignore the mangle table. So for packet filtering firewalls the flowchart in figure 2 can be simplified to the much smaller chart in figure 3.

SAMPLE FIREWALL SCRIPT

To finish off the lesson, here is a sample firewall script for the network and firewall shown in Figure 4. Each line has already been explained in this lesson. You should read through the firewall script and look back through the lesson to make sure you know what it does.

```
iptables -F
iptables -t nat -F
iptables -P FORWARD DROP
iptables -P INPUT DROP
iptables -P OUTPUT DROP
iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE
iptables -A FORWARD -s 192.168.0.0/24 -d 192.168.1.0/24 -j ACCEPT
iptables -A FORWARD -s 192.168.1.0/24 -d 192.168.0.0/24 -j ACCEPT
iptables -A FORWARD -s 192.168.0.0/24 -o ppp0 -j ACCEPT
iptables -A FORWARD -s 192.168.1.0/24 -o ppp0 -j ACCEPT
iptables -A FORWARD -m state –state NEW -i eth0 -j ACCEPT
iptables -A FORWARD -m state –state NEW -i wlan0 -j ACCEPT
iptables -A FORWARD -m state –state ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -m state –state NEW,INVALID -i eth1 -j DROP
iptables -A FORWARD -p udp –-sport 137:139 -j DROP
```

IPTABLES-SAVE & IPTABLES-RESTORE

Once you have configured your firewall, it would be nice to have a simple way to save and restore the ruleset. Red Hat Linux includes two simple utilities which can be used to save and restore the rules: **`iptables-save`** and **`iptables-restore`**.

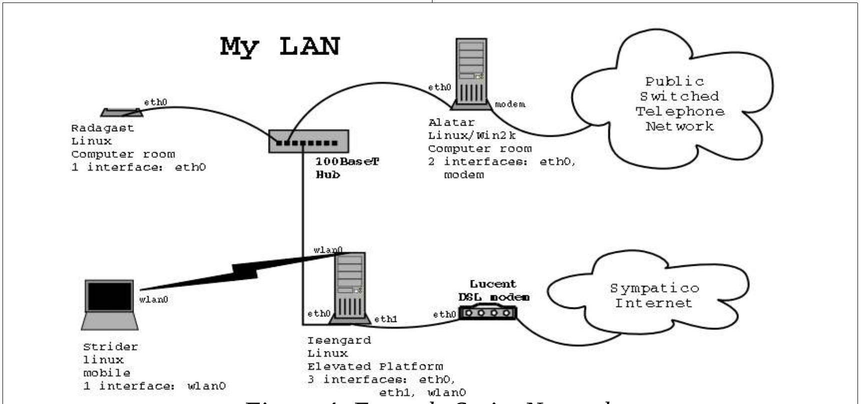

Figure 4: Example Script Network

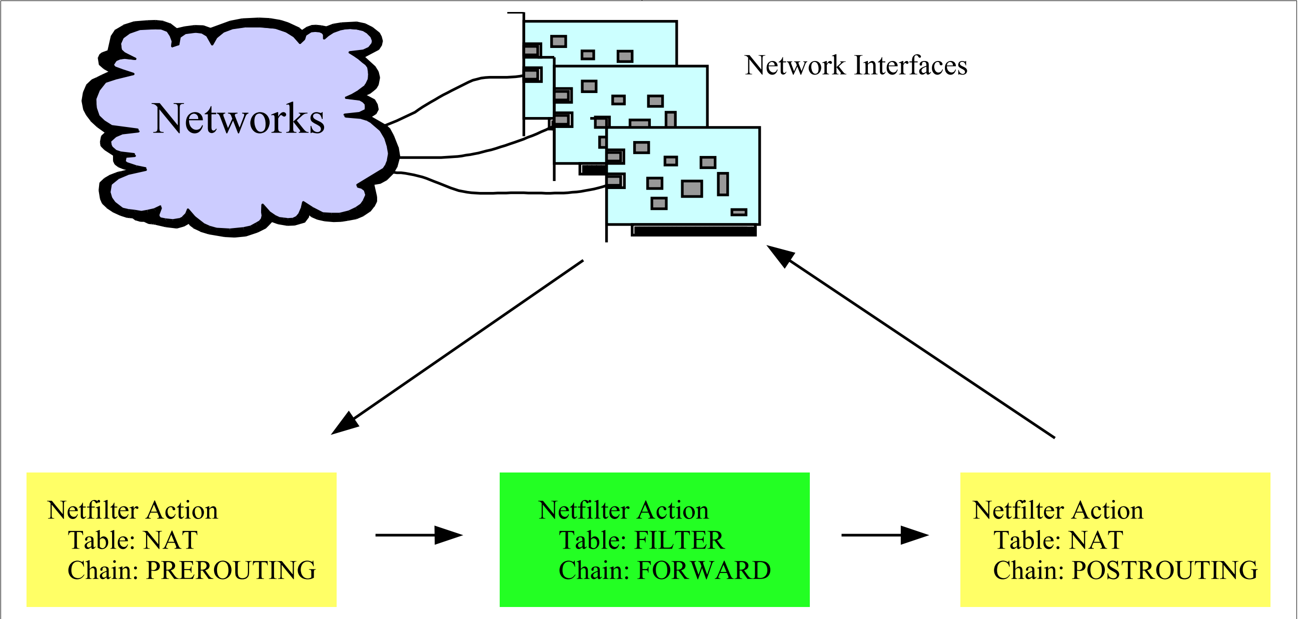

Figure 3: Packet Filtering Firewall Flowchart

`Iptables-save` dumps the contents of the tables to stdout. The format is not specified anywhere, but looks vaguely like the usage of iptables. Here is the iptables-save output after implementing the sample firewall script of the previous section:

```
[root@Radagast andrew]# /sbin/iptables-save
# Generated by iptables-save v1.2.6a on Mon Apr 28 12:47:14 2003
*nat
:PREROUTING ACCEPT [2:96]
:POSTROUTING ACCEPT [4:364]
:OUTPUT ACCEPT [145:9301]
-A POSTROUTING -o eth1 -j MASQUERADE
COMMIT
# Completed on Mon Apr 28 12:47:14 2003
# Generated by iptables-save v1.2.6a on Mon Apr 28 12:47:14 2003
*filter
:INPUT DROP [39:13970]
:FORWARD DROP [0:0]
:OUTPUT DROP [170:17598]
-A FORWARD -s 192.168.0.0/255.255.255.0 -d 192.168.1.0/
255.255.255.0 -j ACCEPT
-A FORWARD -s 192.168.1.0/255.255.255.0 -d 192.168.0.0/
255.255.255.0 -j ACCEPT
-A FORWARD -s 192.168.0.0/255.255.255.0 -o ppp0 -j ACCEPT
-A FORWARD -s 192.168.1.0/255.255.255.0 -o ppp0 -j ACCEPT
-A FORWARD -i eth0 -m state --state NEW -j ACCEPT
-A FORWARD -i wlan0 -m state --state NEW -j ACCEPT
-A FORWARD -m state --state RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i eth1 -m state --state INVALID,NEW -j DROP
-A FORWARD -p udp -m udp --sport 137:139 -j DROP
COMMIT
# Completed on Mon Apr 28 12:47:14 2003
```

This output can be redirected to a file. Later, the file can be used as input to the iptables-restore command using redirection, like this:

```
# /sbin/iptables-save > firewall_rules.save
# /sbin/iptables -F
# /sbin/iptables-restore < firewall_rules.save
```

RED HAT SECURITY LEVEL CONFIGURATION

Red Hat Linux 8 includes a GUI tool called “Security Level Configuration”. This tool provide three preset security levels: none, medium and high. You may also customize the security by accepting communications to specific ports on specific interfaces. When implemented, the tool creates a simple set of rules and used the **`iptables-save`** command to save the rules to the **`/etc/sysconfig/iptables`** file. When you start your computer, the rules will automatically be loaded using the **`iptables-restore`** command. Here is a copy of the **`/etc/sysconfig/iptables`** file generated with security level “high”:

```
# Firewall configuration written by lokkit
# Manual customization of this file is not recommended.
# Note: ifup-post will punch the current nameservers through the
#       firewall; such entries will *not* be listed here.
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:RH-Lokkit-0-50-INPUT - [0:0]
-A INPUT -j RH-Lokkit-0-50-INPUT
-A RH-Lokkit-0-50-INPUT -i lo -j ACCEPT
-A RH-Lokkit-0-50-INPUT -p udp -m udp -s 204.101.251.1 \
                               --sport 53 -d 0/0 -j ACCEPT
-A RH-Lokkit-0-50-INPUT -p udp -m udp -s 204.101.251.2
                               --sport 53 -d 0/0 -j ACCEPT
-A RH-Lokkit-0-50-INPUT -p udp -m udp -s 209.226.175.223
                               --sport 53 -d 0/0 -j ACCEPT
-A RH-Lokkit-0-50-INPUT -p tcp -m tcp --syn -j REJECT
-A RH-Lokkit-0-50-INPUT -p udp -m udp -j REJECT
COMMIT
```