Linking Chains

A methodology for developing rules for IP Chains

Daniel Bradley
daniel@dstc.edu.au

Eric Faccer
efaccer@dstc.edu.au

Mark Cross
mcross@dstc.edu.au

Distributed Systems Technology Centre
Level 12, S Block, QUT Gardens Point
Brisbane Qld 4001, Australia

ABSTRACT

This paper describes a methodology for configuring a packet filter, which is one of the components of a firewall system. It takes into consideration non-obvious security nuances of the TCP/IP protocol stack that may be overlooked by system administrators. The methodology uses the TCP/IP protocol suite's layered architecture as the guide for the composition of the packet filter rule set. It uses the IP Chains packet filter to demonstrate a practical example.

1 INTRODUCTION

Due to the large number of vulnerabilities in today's applications and services, network firewalls have become a common security measure. However, if configured incorrectly, what should be a security measure becomes a security liability.

While there are sources of information that describe the designs and architectures of firewalls, few sources describe how to build the rule sets that are critical for firewalls to meet their intended purpose. This paper intends to address this imbalance by describing a methodology that can be used to formulate rules for the IP Chains packet filtering firewall [1].

The remainder of this paper is organised as follows; section two provides an introduction to firewalls, explaining their purpose and detailing the security vulnerabilities that need to be taken into consideration during configuration. Section three gives an overview of the IP Chains packet filter, describing how Linux handles incoming packets, and how IP Chains decides their fate. Section four presents our methodology by producing an IP Chains configuration script that enforces the network traffic policy of an example network. Section five outlines future work to be done.

2 FIREWALLS

The term firewall describes a safety measure that stops fire in one area spreading to another area. In the building industry it is a protective layer that stops a fire from

---

The work reported in this paper has been funded in part by the Co-operative Research Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Industry, Science & Resources).

spreading from one apartment to another; in the automotive industry it is the barrier that shields the passenger compartment from the engine. To generalize the purpose of a firewall is to stop problems from spreading.

A network firewall is a system of components designed to control the flow of network traffic between networks. This system is usually composed of *packet filters*, which make decisions based on the contents of the packet header, and *application proxies*, which act as middlemen between clients and services. In some instances these components are present on the same machine, but more usually they are located on an adjacent network, commonly referred to as a DMZ (demilitarised zone).

The difference between *packet filters* and *application proxies* can best be explained by identifying the different layers of the TCP/IP stack. Figure 1 shows the TCP/IP stack compared against the OSI model [2]. Each layer encapsulates the previous layer, adding layer relevant information as a header.

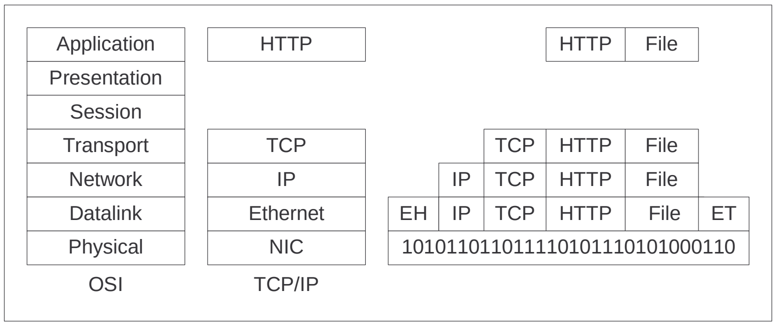

Figure 1 - Network layers

Application proxies work at the application layer and typically act as a man-in-the-middle, terminating the connection to the client and establishing another to the server. When the proxy server receives a request from a client, it may check its validity before making the request to the server on the client's behalf. No actual IP packets are passed between the client and server. The trade-off is that application proxies are typically limited to one particular protocol, e.g. http. An example of an application proxy system is the "TIS Firewall Toolkit" [3].

Packet filters work at the transport layer, either dropping or forwarding packets, by inspecting their network and transport headers.

Configuring a packet filter is a non-trivial task. Below we discuss some non-obvious aspects of TCP/IP communication that need to be checked by all packet filters.

2.1 What a packet filter should filter

When the IPv4 family of protocols were developed security was not an issue, however, subsequent widespread deployment of these protocols has created an environment in which security is an issue. Below we discuss several security weaknesses that need to be addressed when configuring a firewall.

2.1.1 Malformed packets

TCP/IP implementations were designed to expect packets that meet the published TCP/IP specification. Malformed packets do not. When a host can only process specification compliant packets, and it receives a non-conformant packet, the behaviour is undefined.

Two common ways for packets to be malformed are for the TCP flags to be set incorrectly, e.g., two of the SYN, ACK, or FIN, flags set at once; and for the packet length to be different from the actual length. A description of the different ways, intentional and unintentional, that packets have been malformed can be found in [4].

An example of an attack using malformed packets is the attack known as the "Ping of Death" [5], which sends an ICMP *echo* request message that is larger than expected. This causes the packet buffer to overflow, usually causing the machine to crash [6].

The IP Chains system does not provide a way to test whether packets are malformed in any way, and must rely upon a sanity check that is performed when packets are received by the system.

2.1.2 Spoofed source traffic

Many applications use a packet's source address for some degree of authentication, e.g. NFS, and TCP wrappers. By spoofing the address (planting another address in the packet's source address field) it is possible to abuse these trust relationships.

Non-connection oriented protocols such as ICMP and UDP are more susceptible to these attacks due to their lack of a handshake protocol, which requires communication in both directions before a connection is established. An example is the *chargen* attack, where a loop is created between a *chargen* service and an *echo* service, producing a packet storm [7].

A problem with trying to establish a TCP connection using a spoofed source address is that the reply goes to the spoofed host, not to the actual source. Therefore to receive the reply packet it is necessary for the attacker to either; guess the reply, sniff the packet from the network, or perform some form of RARP poisoning [8]. Despite these difficulties, attacks have been successfully carried out using a spoofed TCP connection [9].

It is recommended that at the very least packet filters drop spoofed traffic by performing *ingress* filtering, i.e., dropping any packets that arrive on an interface they shouldn't. This protects the network from external attacks using spoofed packets and from being used as a base for attacks [10].

The *nmap* [11] utility may be used to determine if your routers will pass spoofed packets. The following command uses the –S argument that spoofs the source address on packets sent.

```
nmap <target IP> -S <source IP> -e eth0 –r –sS
```

A useful feature, when wanting to intercept a reply packet to a spoofed address, is the Internet protocol's *source routing* option. This allows the sender to dictate the route that a reply packet travels over the Internet, allowing them to route the packet through a network they control; where it can be conveniently sniffed. Therefore, it is recommended that source routing be disabled; although this may interfere with some mobile IP schemes.

2.1.3 Broadcast traffic

Broadcast addresses are used when a host wants to send a packet to every host on a particular network. This is usually when a host does not know who to contact for some service, or is trying to broadcast information about itself. A common use of broadcasting is the DHCP (Dynamic Host Configuration Protocol) [12], which sends a packet to the broadcast address and receives a reply in a similar fashion. Directed broadcasts are broadcasts targeted at a particular network.

The misuse of broadcast traffic has been used in attacks such as the Smurf attack [13]. In this attack a series of spoofed ICMP *echo* request packets are sent to a network's broadcast address. If the router allows *directed broadcasts*, these requests will be forwarded on to each host on that network. The result of this is that the host at the address that was spoofed can be deluged with ICMP echo response packets.

Another related issue is that if broadcast packets are not specifically denied or allowed, they tend to be logged; this may obscure more important log messages.

2.1.4 Improper traffic

The Internet protocol address range is broken up into a number of different classes. Only four of these are intended to be routable across the public Internet, one of which is only used for multicast traffic. The others are reserved for future use.

In general an Internet router should not route any traffic containing addresses belonging to the *private* address ranges [14], or any addresses that match the address mask "240.0.0.0/5", which are reserved and undefined addresses.

2.1.5 ICMP and IGMP packets

ICMP (Internet Control and Management Protocol) and IGMP (Internet Group Management Protocol) packets are used for sending control and management information about connections. They can be used to perpetrate denial of service attacks such as the ping of death [15]. ICMP can also be abused to allow scanning of a network [16]. Possible information that can be acquired from such a scan includes remote operating system type, remote network topology, and the existence of a remote host, this is called enumeration.

ICMP messages can be grouped into two classes: error messages, and query messages. Many of the query messages are now deemed obsolete by specification and should be blocked. For example, *information request* & *information reply* (types 15,16) messages.

Of the error messages, the *destination unreachable* messages (type 3) are the most important for network performance. These prevent lengthy timeouts when a host is unable to respond [17]. These messages are also required for path MTU discovery, a process important for optimising bandwidth usage and preventing network degradation.

Another error message, *source quench* messages (type 4) are required for transport layer flow control. Other ICMP messages can be blocked safely without adversely affecting network performance, but doing so will lead to loss of certain types of functionality. For example, the ***ping*** tool relies on being able to send *echo* messages (type 8) and receive *echo reply* messages (type 0). The ***trace route*** tool uses *time exceeded* (type 11) messages to view a datagram's route through a network. This gives information about the network's logical topology and is useful for detecting routing loops. This information, useful for a system administrator is also potentially useful for an attacker building a detailed view of a network, and as such should be carefully considered in a network security policy.

However, in certain cases, allowing certain messages (types 8,0,11) may be deemed acceptable due to mitigating factors. For example, it could be considered futile to block *echo* messages to prevent enumeration of a networked host if that server is vulnerable to higher protocol layer scan technique such as TCP port SYN scanning. Thus, blocking ICMP messages to prevent network enumeration should be reserved for workstations that do not listen on ports.

2.1.6 Local router traffic

Due to the crucial role of the packet filter - enforcing an organisation's network traffic policy - it is necessary that it be heavily protected. This is achieved by ensuring that there are few, if any, services running on the machine, that only those services are allowed to accept packets, and only certain machines are allowed to connect to them.

In most cases there is only a single service for remote administration. With Linux based firewalls this is usually the *secure shell* daemon. It is easy, however, to make simple mistakes that could allow external access to this service. In particular an assumption that a person might make is that if a service is bound to the internal interfaces' address then only internal machines will be able to connect to the service. This is not true, as, by default some routers will route to any of its interfaces [18].

3 IP CHAINS

IP Chains is a packet filter for Linux systems. A packet filter compares the fields of each packet's header against a set of rules; these rules determine the fate of the packet, i.e., allowed, or dropped. Below is a brief overview of the IP Chains system. For a more in-depth explanation consult the Linux Documentation Project's IP Chains HOWTO [1].

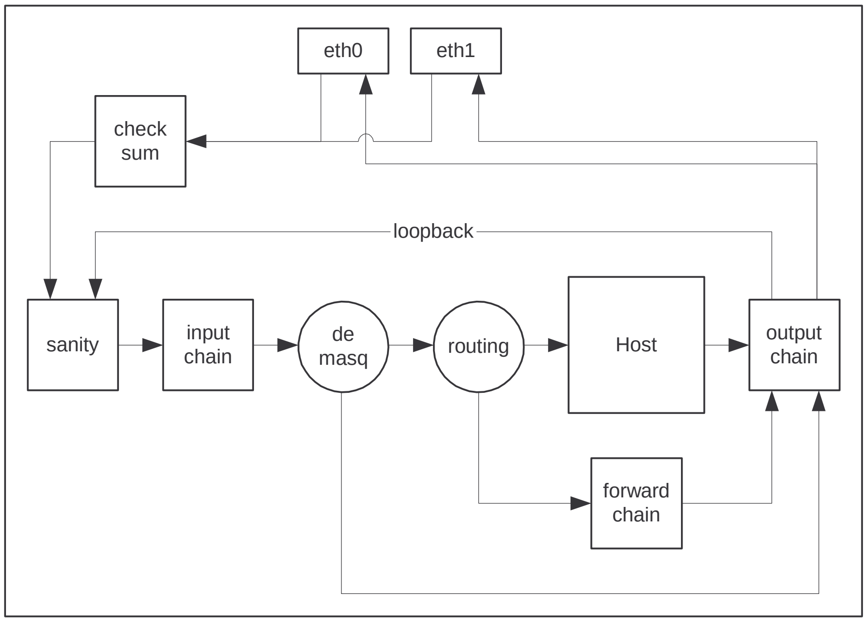

Figure 2 - How packets traverse the filters

3.1 Chains

The IP Chains system uses lists of rules called chains. There are three built-in chains; **input**, **forward** and **output**; which are used for incoming, forwarded and outgoing traffic, respectively. It is also possible to create custom chains that may be jumped to from any of the built-in chains.

Figure 2 shows how incoming packets are handled by a Linux system. After having their checksum verified and passing a sanity check, all packets are processed by the **input** chain.

If they are accepted; masqueraded packets are passed directly to the output chain; local packets are passed to the system; and forward packets are passed to the **forward** chain and then to the **output** chain.

When a packet is handed to a chain for processing, it is compared progressively against each rule in that chain. Each rule contains a *target* that specifies what is to be done to the packet if the rule is matched - common targets are **ACCEPT**, **REJECT** and **DENY**. The target may also be a custom chain, which causes the packet to be compared against each rule in that chain until a further match is made.

If a packet reaches the end of a custom chain processing returns to previous chain. If the packet reaches the end of a built-in chain, its fate is determined by that chain's *policy*, which will be to either: accept, reject or deny.

3.2 Rules

In IP Chains each rule specifies criteria a packet must match, and a target to jump to if that criteria is matched. The possible targets are: ACCEPT, the packet is either delivered to the system, or to either the **forward** or **output** chain, as appropriate; REJECT, the packet is dropped and an ICMP message is returned; DENY, the packet is dropped; MASQ, the packet is masqueraded as from this host; REDIRECT, the packet is redirected to a local port; RETURN, processing returns to the previous chain. The target may also be a custom chain.

The criteria to be matched are based upon the attributes of a network packets header. What can be matched depends upon the type of the packet: ICMP, UDP, or TCP. All packets can be matched on their source and destination addresses, as well as the protocol type of the packet. ICMP packets can be matched against the ICMP message type, while UDP and TCP packets can be matched against their source and destination ports. TCP packets can also be matched against whether the SYN flag is set.

3.3 Commands

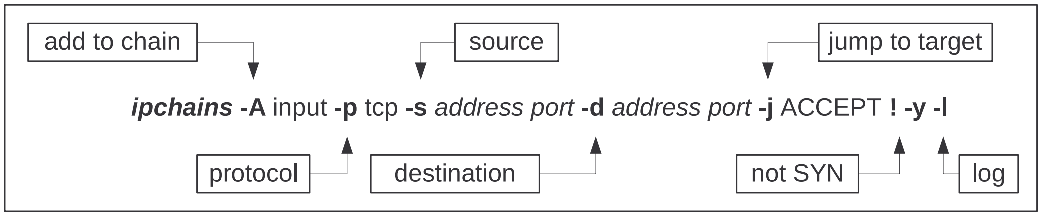
add to chain
source
jump to target
ipchains -A input -p tcp -s address port -d address port -j ACCEPT ! -y -l
protocol
destination
not SYN
log

Figure 3 - IP Chains command

In IP Chains, rules are added to chains, and new custom chains are added, using the ***ipchains*** command. The specification of the command is as follows:

```
ipchains -[ADC] chain rule-specification [options]
```

Where “chain” is the chain the rule is to be added to, “rule-specification” specifies the criteria that must be matched, and “[options]” are various options that can be used, e.g., verbose mode (“-v”).

Figure 3 shows an example of an IP Chains command. This command would add a rule to the **input** chain, which matches against a TCP packet that contains the source and destination address specified, and that is not a packet initiating a connection - does not have the SYN flag set. A matching packet would be logged, before continuing on.

4 METHODOLOGY

This section gives a step-by-step description of the methodology and produces a useable configuration script for an IP Chains firewall. This script should be run during system initialisation before any of the network interfaces are brought up.

4.1 Overview

When it is run, the script creates several custom chains, which along with the built-in chains, are then populated with rules. Network packets are mostly accepted or denied in the custom chains, with the rules in the built-in chains passing them to the custom chains for processing.

Figure 4 shows how incoming traffic flows through the chains. All incoming traffic is passed to the **input** chain. Each packet is then passed to each of the special chains, beginning with the **ingress** chain, until it is either denied or deemed to be valid. If the packet is destined locally, it is then passed to one of the local chains (one for each interface) until it is accepted; else it is logged and denied. Packets to be forwarded pass through these rules unmatched and are then accepted by forward rules, causing them to be passed to the **forward** chain.

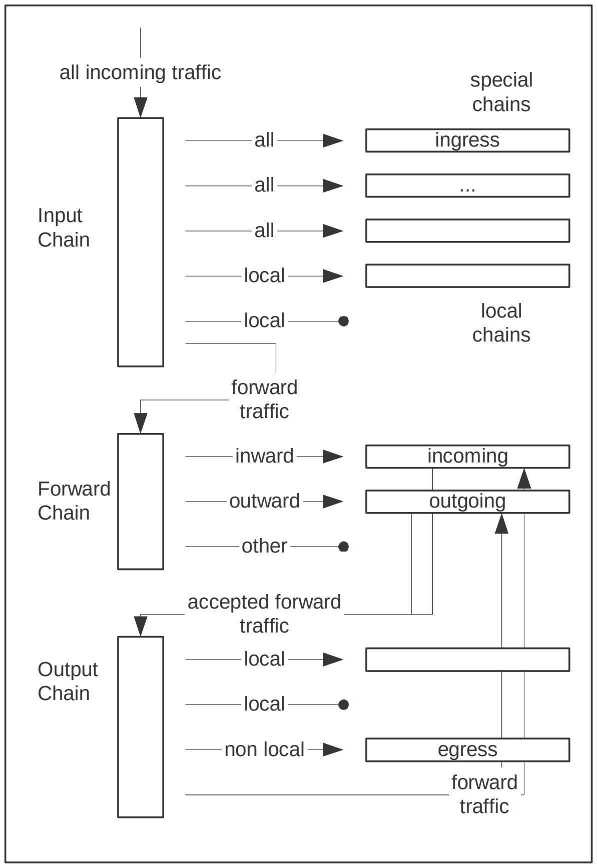

Figure 4 - Graphical representation of final state

When packets are received by the **forward** chain they are passed to either an incoming or outgoing chain as appropriate. These chains contain rules that match those packets that are to be forwarded by the packet filter; those that aren't are denied and logged.

In the **output** chain, packets that claim to be from the local host are processed first; sent to an appropriate chain for each interface, and if not accepted are denied and logged. Any remaining packets should be those that came from the **forward** chain. They are first sent to a chain that perform egress filtering, and then are double-checked by being resent to either the incoming or outgoing chain.

4.2 Example network and policy

The following example network is typical of those of organizational units or small businesses. An internal network is connected to an external public network; a properly configured firewall is required to control the traffic between the two networks. A good resource for determining appropriate rules for other protocols is the text "Building Internet Firewalls" [19].

4.2.1 The network

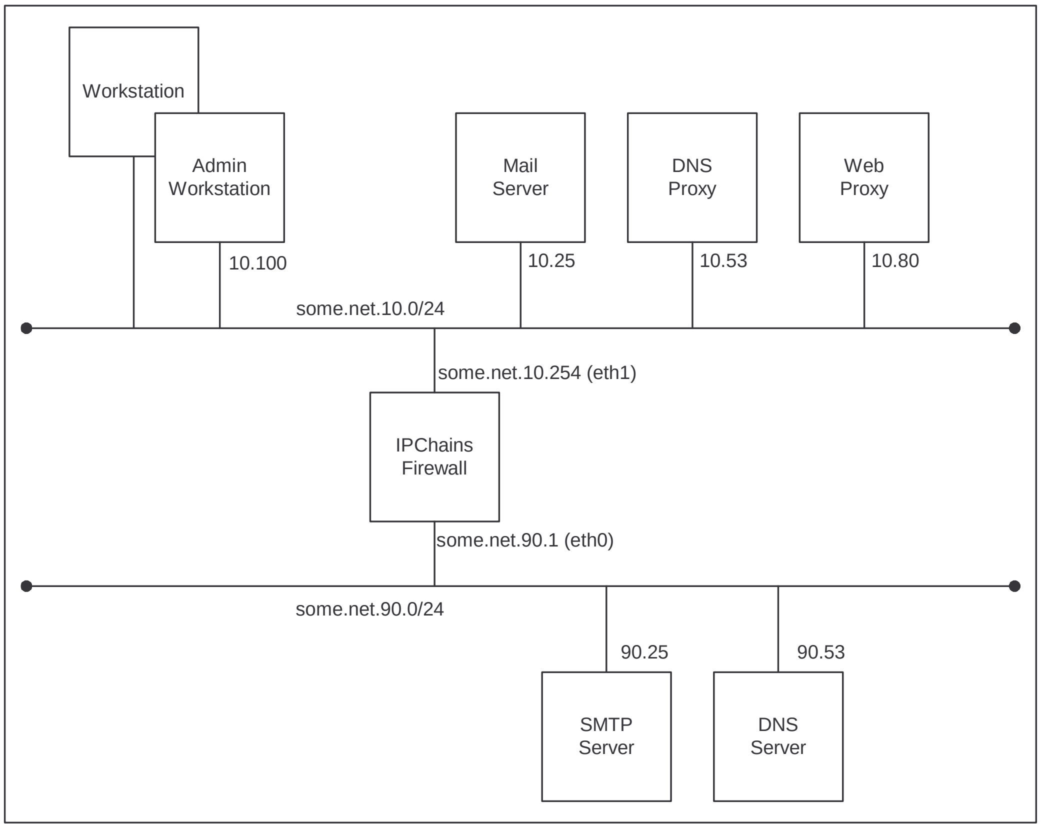

Figure 5 - The example network

The internal network contains a variety of machines including workstations, servers, and proxy servers. The only workstation that affects the firewall configuration is an administrative workstation used to perform remote administration of the packet filter. There are three relevant servers: a mail server, a DNS proxy, and a web proxy. We'll assume that these have been assigned identifiers corresponding the service they provide, e.g., the web proxy's address is `some.net.10.80`. The external network contains an SMTP server and a DNS server.

4.2.2 The network security policy

A network traffic policy should specify what traffic is allowed to flow within, into and out from, an organisational network. For brevity's sake the policy below only covers the traffic that contacts or crosses the firewall.

The network traffic policy for traffic from/to the firewall is as follows:

1. The firewall may send ICMP type 11 (time-exceeded) messages to the internal network.
2. The firewall may receive ICMP type 8 (echo) messages from the internal network.
3. The firewall may send ICMP type 0 (echo reply) messages to the internal network.
4. The firewall may send DNS packets to, and received DNS packets from, the internal DNS proxy.

5. The firewall may accept SSH connections from the Admin Workstation.

The network traffic policy for traffic over the firewall is as follows:

6. The firewall will forward incoming and outgoing ICMP type 3 (destination-unreachable) messages.
7. The firewall will forward incoming ICMP type 11 (time-exceeded) messages.
8. The firewall will forward incoming ICMP type 0 (echo reply) messages.
9. The firewall will forward outgoing ICMP type 8 (echo) messages.
10. The DNS proxy may send UDP datagrams to, and receive reply DNS datagrams from, the external DNS server.
11. The Web Proxy may make connections using http to any external web server.
12. The Mail Server may make SMTP connections to, and receive SMTP connections from, the external SMTP server.
13. All internal workstations may make SSH connections to any external machine.

4.3 Step-by-step

The methodology that we demonstrate below makes the following assumptions: the firewall has two network interfaces – no more, no less; one of these interfaces is connected to an external network and the other is connected to an internal network; and both of these have been assigned valid public class C subnet addresses.

The process consists of preparation; which includes setting kernel parameters, defining shell variables, and setting default policies; creation of custom chains; addition of special rules, which filter various types of improper traffic; addition of accept rules, which allow the traffic that is intended to be allowed; and then lastly the addition of "glue" rules, which pass the packets to appropriate chains for processing.

As you read, it should be assumed that each command given is concatenated onto the configuration script, giving at the end the complete script. We place this script in the "/etc/rc.d" directory and give it the name "rc.ipchains". It is set to execute during system boot up before any network interfaces are brought up.

4.3.1 Enabling IP Chains

Firstly IP Chains needs to be enabled. IP Chains is composed of two parts: the kernel functionality that processes the packets, and the administration program *ipchains*. If IP Chains has been compiled as a kernel module it may be necessary to load the module into the kernel. Loading the module can be achieved by adding the following command to the configuration script:

```
/sbin/insmod ipchains
```

It is not the intention of this paper to describe how to do the installation of IP Chains; if you need further information, in this regard, you should consult the Linux Documentation Project's IP Chains HOWTO [1] and Firewalling HOWTO [20].

4.3.2 Setting Linux kernel parameters

There are some Linux kernel parameters that may be set to increase the effectiveness of the packet filter [21]. Most important, is enabling IP forwarding.

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

The following will cause a packet to be dropped if a reply would leave through a different interface to the one it was received on.

```
for f in /proc/sys/net/ipv4/conf/*/rp_filter; do
  echo 1 > $f
done
```

Source routing may be disabled.

```
for f in /proc/sys/net/ipv4/conf/*/accept_source_route; do
  echo 0 > $f
done
```

Also, directed broadcasts can be disabled.

```
echo 1 > icmp_echo_ignore_broadcasts
```

4.3.3 Interface information

To make later commands easier to read and maintain, information about each interface is stored in shell variables.

```
#  Public interface information

ETH0=eth0
ETH0_MASK=255.255.255.0
ETH0_NETID=<external net id>              ## eg. 192.168.33
ETH0_IP=$ETH0_NETID.1
ETH0_NET=$ETH0_NETID.0/$ETH0_MASK
ETH0_BCAST=$ETH0_NETID.255

#  Private interface information

ETH1=eth1
ETH1_MASK=255.255.255.0
ETH1_NETID=<internal net id>              ## eg. 192.168.33
ETH1_IP=$ETH1_NETID.254
ETH1_NET=$ETH1_NETID.0/$ETH0_MASK
ETH1_BCAST=$ETH1_NETID.255
```

We will also define the following variables.

```
PUBLIC=”! $ETH1_NET”
IPCHNS=/sbin/ipchains
```

In particular, note that **ETH*x*_NETID** only represents the first 3 octets of an network address. Hence $**ETH0_NETID**.255 would be the broadcast address of the first interface.

4.3.4 Setting default chain policies

The policies of the built-in chains should be set to a safe default. This ensures that if there is a problem with the rest of the configuration that packets will be dropped by default.

```
$IPCHNS –P input   DENY
$IPCHNS –P forward DENY
$IPCHNS –P output  DENY
```

4.3.5 Flush existing rules

We now flush existing rules.

```
$IPCHNS --flush
```

4.3.6 Create custom chains

Here we create the chains we presented in figure 4 that we will be populating. Firstly we create the chains for the special rules.

```
$IPCHNS –N ingress     2> /dev/null
$IPCHNS –N bcast       2> /dev/null
$IPCHNS –N undef       2> /dev/null
$IPCHNS –N reserved    2> /dev/null
$IPCHNS –N mcast       2> /dev/null
$IPCHNS –N private     2> /dev/null
$IPCHNS –N egress      2> /dev/null
```

A chain is created for incoming and outgoing local traffic on each interface.

```
$IPCHNS –N inLO        2> /dev/null
$IPCHNS –N in$ETH0     2> /dev/null
$IPCHNS –N in$ETH1     2> /dev/null
$IPCHNS –N outLO       2> /dev/null
$IPCHNS –N out$ETH0    2> /dev/null
$IPCHNS –N out$ETH1    2> /dev/null
```

Also a chain is created for each direction of forwarded traffic.

```
$IPCHNS –N $ETH0$ETH1  2> /dev/null
$IPCHNS –N $ETH1$ETH0  2> /dev/null
```

As flushing IP Chains using **--flush** doesn’t delete custom created chains, possible error output, from creating an already existing chains is send to **/dev/null**. Note that chain names have a maximum length of 8 characters, thus if you give a chain a name longer than this, attempting to add a rule will cause errors.

4.3.7 Special chains rules

Firstly, the rules that protected against malicious packets are added to the special chains.

4.3.7.1 Ingress

The following rules perform ingress filtering. This rule protects against traffic that pretends to be from the loopback address, but isn’t.

```
$IPCHNS –A ingress –i ! lo –s 127.0.0.0/24 –j DENY -l
```

Next a rule is added for each network interface. The first rule denies and logs packets that were received on the external interface but claim to be from the internal network; the second denies and logs packets that were received on the internal interface but claim to be from a public address.

```
$IPCHNS –A ingress –i $ETH0 –s $ETH1_NET –j DENY -l
$IPCHNS –A ingress –i $ETH1 –s $PUBLIC   –j DENY -l
```

4.3.7.2 Broadcast

The following rules prevent broadcast packets from being forwarded or accepted by the firewall. The first two rules deny and log packets that are using the broadcast addresses improperly, i.e., using the destination address as the source address and vice versa.

```
$IPCHNS –A bcast -s 255.255.255.255 –j DENY -l
```

```
$IPCHNS –A bcast –d 0.0.0.0         -j DENY -l
```

The following rules simply deny all broadcast packets. Due to the difficulty of differentiating between valid and invalid broadcast traffic, these packets are not logged.

```
$IPCHNS –A bcast –s 0.0.0.0         -j DENY
$IPCHNS –A bcast –d 255.255.255.255 –j DENY
$IPCHNS –A bcast –d $ETH0_NETID.0   –j DENY
$IPCHNS –A bcast –d $ETH0_NETID.255 –j DENY
$IPCHNS –A bcast –d $ETH1_NETID.0   -j DENY
$IPCHNS –A bcast –d $ETH1_NETID.255 –j DENY
```

We also want to disable loop-back broadcast traffic.

```
$IPCHNS –A bcast –d 127.0.0.0       –j DENY
$IPCHNS –A bcast –d 127.255.255.255 –j DENY
```

4.3.7.3 Undefined, Reserved and Multicast addresses

The following commands protect against routing packets that contain non-routable addresses. These rules are added to the **undef**, **reserved**, and **mcast** chains.

Later, when the packets are process by these chains in the **input** chain, order is important, as the mask used for multicast traffic also matches addresses matched by the masks for reserved and undefined addresses. It would be possible to combine all these rules into one rule, however leaving them as separate rules in separate chains makes their log entries easier to identify. Receiving packets containing undefined addresses is more likely to be of concern than receiving multicast packets.

First we add the rules to detect undefined addresses.

```
$IPCHNS –A undef –s 248.0.0.0/5 –j DENY –l
$IPCHNS –A undef –d 248.0.0.0/5 –j DENY -l
```

Next we add the reserved rules.

```
$IPCHNS –A reserved –s 240.0.0.0/5 –j DENY –l
$IPCHNS –A reserved –d 240.0.0.0/5 –j DENY –l
```

Lastly we add the multicast rules.

```
$IPCHNS –A mcast –s 224.0.0.0/4 –j DENY
$IPCHNS –A mcast –d 224.0.0.0/4 –j DENY
```

4.3.7.4 Private Address Ranges

These rules block traffic to and from private subnets.

```
$IPCHNS –A private –s 10.0.0.0/8     –j DENY -l
$IPCHNS –A private –d 10.0.0.0/8     –j DENY –l
$IPCHNS –A private –s 169.254.0.0/16 –j DENY –l
$IPCHNS –A private –d 169.254.0.0/16 –j DENY –l
$IPCHNS –A private –s 172.16.0.0/12  –j DENY -l
$IPCHNS –A private –d 172.16.0.0/12  –j DENY –l
$IPCHNS –A private –s 192.0.2.0/24   -j DENY –l
$IPCHNS –A private –d 192.0.2.0/24   -j DENY -l
$IPCHNS –A private –s 192.168.0.0/16 –j DENY -l
$IPCHNS –A private –d 192.168.0.0/16 –j DENY -l
```

4.3.8 Allowing Services

We now add the rules that implement the policy set forth in the network security policy.

4.3.8.1 Local Services

Rules for local services are added to the chains with the ***in*** and ***out*** prefix. For each incoming connection a connection rule is added to the appropriate *in* chain, and a reply rule is added to the appropriate *out* chain. Similarly the reverse happens for each outgoing connection.

To implement policy number (1), which allows ICMP time-exceeded messages to the internal network.

```
$IPCHNS –A out$ETH1 –p icmp –-icmp-type 11 –d $ETH1_NET –j ACCEPT
```

To implement policy number (2), which allows ICMP echo messages from the internal network.

```
$IPCHNS –A in$ETH1 –p icmp --icmp-type 8 –s $ETH1_NET –j ACCEPT
```

To implement policy number (3), which allows ICMP echo reply messages from the firewall to the internal network.

```
$IPCHNS –A out$ETH1 –p icmp --icmp-type 0 –d $ETH1_NET –j ACCEPT
```

To implement policy number (4), which allows the firewall to make DNS requests, the following rules are added to the **out$ETH1** and **in$ETH1** chains.

```
$IPCHNS –A out$ETH1 \
  –p udp --sport domain -d $ETH1_NETID.53 domain –j ACCEPT
$IPCHNS –A in$ETH1 \
  –p udp –s $ETH1_NETID.53 domain --dport domain –j ACCEPT
```

To implement policy number (5), which allows an administrative workstation to connect to the *ssh* service, the following rules are added to the **in$ETH1** and **out$ETH1** chains respectively.

```
$IPCHNS –A in$ETH1 \
  –p tcp -s $ETH1_NETID.100 --dport ssh –j ACCEPT
$IPCHNS –A out$ETH1 \
  –p tcp --sport ssh –d $ETH1_NETID.100 –j ACCEPT ! –y
```

4.3.8.2 Forwarded Services

For packets that are to be forwarded from one interface to the other, rules are added to either the **$ETH1$ETH0** or **$ETH0$ETH1** chain, as is appropriate.

To implement policy number (6), which allows incoming and outgoing ICMP destination-unreachable messages.

```
$IPCHNS –A $ETH0$ETH1 –p icmp --icmp-type 3 -j ACCEPT
$IPCHNS –A $ETH1$ETH0 –p icmp --icmp-type 3 -j ACCEPT
```

To implement policy number (7), which allows incoming ICMP time-exceeded messages.

```
$IPCHNS -A $ETH0$ETH1 -p icmp --icmp-type 11 -j ACCEPT
```

To implement policy number (8), which allows incoming ICMP echo reply messages.

```
$IPCHNS -A $ETH0$ETH1 -p icmp --icmp-type 0 -j ACCEPT
```

To implement policy number (9), which allows outgoing ICMP echo messages.

```
$IPCHNS -A $ETH1$ETH0 -p icmp --icmp-type 8 -j ACCEPT
```

To implement policy number (10) allow DNS we add.

```
$IPCHNS –A $ETH1$ETH0 –p udp \
  –s $ETH1_NETID.53 domain –d $ETH0_NETID.53 domain –j ACCEPT
$IPCHNS –A $ETH0$ETH1 –p udp \
  –s $ETH0_NETID.53 domain –d $ETH1_NETID.53 domain –j ACCEPT
```

To implement policy number (11) allowing HTTP traffic we add.

```
$IPCHNS –A $ETH1$ETH0 \
  –p tcp –s $ETH1_NETID.80 3128 –d $PUBLIC http –j ACCEPT
$IPCHNS –A $ETH0$ETH1 \
  –p tcp –s $PUBLIC http –d $ETH1_NETID.80 3128 –j ACCEPT ! –y
```

To implement policy number (12) allowing SMTP connections we add the following rules.

First we add the rules to allow outgoing connections to the external SMTP server.

```
$IPCHNS –A $ETH1$ETH0 -p tcp –s $ETH1_NETID.25 \
                             –d $ETH0_NETID.25 smtp –j ACCEPT
$IPCHNS –A $ETH0$ETH1 -p tcp -s $ETH0_NETID.25 smtp \
                             –d $ETH1_NETID.25 –j ACCEPT ! –y
```

Then we add rules to allow the external SMTP server to connect to our internal one.

```
$IPCHNS –A $ETH0$ETH1 -p tcp –s $ETH0_NETID.25 \
                      –d $ETH1_NETID.25 smtp –j ACCEPT
$IPCHNS –A $ETH1$ETH0 –p tcp -s $ETH1_NETID.25 smtp \
                             -d $ETH0_NETID.25 –j ACCEPT ! –y
```

Lastly to implement policy number (13), allowing ssh connections to outside hosts we add.

```
$IPCHNS –A $ETH1$ETH0 -p tcp –s $ETH1_NET
                             –d $PUBLIC ssh –j ACCEPT
$IPCHNS –A $ETH0$ETH1 -p tcp –s $PUBLIC ssh
                             –d $ETH1_NET –j ACCEPT ! -y
```

4.3.9 Egress Filtering

The reason for egress filtering is to ensure that outgoing packets actually came from where they claim to have come from. This seeks to protect organizations from being used as staging areas for launching attacks against other systems.

Egress filtering is a variation of ingress filtering, which has already been performed in the **ingress** chain. The only case that has not been allowed for is when the firewall itself sends spoofed packets. On first thought one might think that if something is causing the firewall to send spoofed packets then chances are that it would also have sufficient privileges to remove any firewall rules intended to stop this. This is probably true, but assuming the possibility that it could be a worm sending the packets the following rules will attempt to stop them.

Note that packets originating from the firewall that have a valid source address and will leave out an appropriate interface have already been accepted at this stage.

The first rule stops the firewall from sending a packet to a external host with the spoofed source address of an external host. The second rule stops the firewall from sending a packet to an internal host with the spoofed source address of an external host.

```
$IPCHNS –A egress –i $ETH1 –s $ETH1_NET –j DENY –l
$IPCHNS –A egress –i $ETH0 –s $PUBLIC   –j DENY -l
```

Unfortunately, because it is not possible in IP Chains to differentiate between a packet that has been passed from the forward chain, and a packet that has originated from the local host, it is not possible to protect against the firewall sending a packet to an external address with the spoofed source address of an internal address.

One reason an attacker may do this is to hide the fact that the firewall has been compromised.

4.3.10 Linking the chains

So we currently have the following chains; the special chains, **ingress**, **bcast**, **mcast**, **private**, and **egress**; the local chains, **in$ETH0**, **out$ETH0**, **in$ETH1**, and **out$ETH1**; and the forward chains, **$ETH0$ETH1**, and **$ETH1$ETH0**. But at this stage no packets actually reach these chains, as the default built-in chains **input**, **forward**, and **output**, are currently empty.

It is necessary to add rules to these chains that pass the packets to the other chains for filtering. Also the ordering is important as well as packets should be processed by the special chains, then the local chains, and only lastly the forward chains.

4.3.11 The input chain

4.3.11.1 Jumping to the special chains

It is first necessary to jump to the special chains. These perform the task of specifically matching illegal, invalid, or unwanted broadcast and multicast packets, then dropping them. All packets must be processed by these six chains before being processed by the local chains.

```
$IPCHNS –A input –j ingress
$IPCHNS –A input –j bcast
$IPCHNS –A input –j undef
$IPCHNS –A input –j reserved
$IPCHNS –A input –j mcast
$IPCHNS –A input –j private
```

4.3.11.2 Handling Local Traffic

Next it is necessary to handle local traffic. This must be processed before traffic to be forwarded, as forwarded traffic may accept services not appropriate for the firewall. And as the firewall itself also has IP addresses that belong to the networks either side of it, it may mistakenly be included in a rule that accepts a connection.

Also if forwarded traffic is processed in the forward chain, then it needs to be accepted in the input chain first. This means that blanket acceptance of packets to cause them to move to the forward chain, will also cause those packets addressed to a local interface to be accepted as well.

Therefore for each local interface, if packets are destined for that IP address they are passed to the relevant local chain, where if they are to be accepted, the will be accepted. Any that aren’t are dropped by the next rule.

Rules for the loop-back interface:

```
$IPCHNS –A input –i lo    –d $ETH0_IP    –j inLO
$IPCHNS –A input –i lo    –d $ETH1_IP    –j inLO
$IPCHNS –A input –i lo    –d 127.0.0.0/8 –j inLO
$IPCHNS –A input –i       –d 127.0.0.0/8 –j DENY -l
```

For the external interface:

```
$IPCHNS –A input –i $ETH0 –d $ETH0_IP –j in$ETH0
$IPCHNS –A input          –d $ETH0_IP –j DENY -l
```

And for the internal inteface:

```
$IPCHNS –A input –i $ETH1 –d $ETH1_IP –j in$ETH1
$IPCHNS –A input          –d $ETH1_IP –j DENY -l
```

For the rule that passes the packet to the custom chain we check the interface to make sure that the packet came in an appropriate interface for that chain.

4.3.11.3 Handling Forwarding Traffic

Next it is necessary to process packets to be forwarded. Forwarded packets need to be accepted by the input chain, which passes them to the forward chain, then accepted by the forward chain, which passes them to the output chain, then accepted by the output chain.

So you need to add a rule to the input chain to match packets to be forwarded. These packets can either be accepted, causing them to be passed to the forward chain, or passed to the appropriate $ETHx$ETHx chain, in which case if they are allowed they will be passed to the forward chain.

```
$IPCHNS –A input –i $ETH0 –d   $ETH1_NET –j ACCEPT
$IPCHNS –A input –i $ETH1 –d ! $ETH1_NET –j ACCEPT
```

4.3.11.4 Deny

At the end of the input chain we add a rule that denies and logs any remaining packets.

```
$IPCHNS -A input -j DENY -l
```

4.3.12 The forward chain

Next we add rules to the forward chain that are similar to those of the **input** chain except that they now pass the packets to a chain appropriate for the direction of traffic. Note that in the **forward** and **output** chains, the '-i' argument means the output interface.

```
$IPCHNS –A forward –s $PUBLIC   –i $ETH1 –j $ETH0$ETH1
$IPCHNS –A forward –s $ETH1_NET –i $ETH0 –j $ETH1$ETH0
```

The ingress filtering means that we can trust the source and destination addresses so we could also use these rules.

```
$IPCHNS –A forward –s $PUBLIC   –d $ETH1_NET –j $ETH0$ETH1
$IPCHNS –A forward –s $ETH1_NET –d $ETH0_NET –j $ETH1$ETH0
```

We also add a rule that denies and logs any remaining traffic.

```
$IPCHNS -A forward -j DENY -l
```

4.3.13 The output chain

With the input chain, all packets can be processed as though they have just been received by the box. However, with the output chain, packets may be coming from either the forward chain, or may have originated from the firewall. For this reason local traffic is processed first. Then packets are sent to the egress chain. Then any forward traffic is accepted.

4.3.13.1 Local traffic

Local traffic is processed first by filtering out traffic sourced from a local address - loopback or local interface. The outgoing interface is checked to only match packets that will be going out an appropriate interface; effectively performing egress filtering on those packets. Inappropriate packets are denied and logged.

```
$IPCHNS -A output -i lo    -s $ETH0_IP    -j outLO
$IPCHNS -A output -i lo    -s $ETH1_IP    -j outLO
$IPCHNS –A output –i lo    –s 127.0.0.0/8 –j outLO
$IPCHNS –A output          –s 127.0.0.0/8 –j DENY -l
$IPCHNS –A output –i $ETH0 –s $ETH0_IP    –j out$ETH0
$IPCHNS –A output          –s $ETH0_IP    –j DENY -l
$IPCHNS –A output –i $ETH1 –s $ETH1_IP    –j out$ETH1
$IPCHNS –A output          –s $ETH1_IP    –j DENY -l
```

4.3.13.2 Egress filtering

Next we perform egress filtering on the remaining outgoing packets - those that have come from the forward chain - by jumping to the **egress** chain.

```
$IPCHNS –A output –j egress
```

4.3.13.3 Forward traffic

At this point any packets should be forward packets; we double check by jumping to the appropriate forward chain.

```
$IPCHNS –A output –s $PUBLIC   –i $ETH1 –j $ETH0$ETH1
$IPCHNS –A output –s $ETH1_NET –i $ETH0 –j $ETH1$ETH0
```

4.3.13.4 Deny

Finally we add a rule that denies and logs any packets that have survived thus far.

```
$IPCHNS -A output -j DENY -l
```

5 FUTURE DIRECTIONS

The methodology presented does not consider the use of private networks, masquerading, or the possibility of connecting more than two networks. These areas will be addressed; and a similar methodology will be produced for the IP Chains successor IP Tables [22].

6 CONCLUSION

While firewall rule sets are incredibly complex, a thorough methodology for developing the rule sets can increase their quality tremendously. We have discussed problem traffic types that need to be considered when developing a firewall configuration and have presented a methodology for developing rules sets that take these into account.

REFERENCES

[1] P. Russel, “*Linux IPCHAINS HOWTO*”. Version 1.0.8, July 2000. http://www.tldp.org/HOWTO/IPCHAINS-HOWTO.html

[2] Standard ISO 7498 “*Information Processing Systems - Open Systems Interconnection - Basic Reference Model*”. Reference Number: ISO 7498:1984(E), ISO TC97/SC21 Secretariat, ANSI, New York, New York, 1984.

[3] M.J. Ranum, F. M. Avolio, “*A Toolkit and Methods for Internet Firewalls*”. In Technical Summer Conference, pages 37--44, Boston, Massachusetts, June 1994. USENIX.

[4] M. Bykova, “*Statistical analysis of malformed packets and their origins in the modern Internet*”. March 2002. http://irg.cs.ohiou.edu/papers/.

[5] CERT Advisory CA-1996-21, “*TCP SYN flooding and IP spoofing attacks*”. http://www.cert.org/advisories/CA-1996-21.html

[6] Ping of death. http://www.insecure.org/sploits/ping-o-death.html

[7] CERT Advisory CA-1996-01, “*UDP port denial-of-service attack*”. http://www.cert.org/advisories/CA-1996-01.html

[8] S. Whalen 2001, “*An introduction to ARP spoofing*”. Revision 1, April, 2001 http://chocobospore.org/projects/arpspoof/arpspoof.pdf

[9] CERT Advisory CA-1995-01, “*IP spoofing attacks and hijacked terminal connections*”. http://www.cert.org/advisories/CA-1995-01.html

[10] P. Ferguson, D. Senie, “*Network ingress filtering: defeating denial of service attacks which employ IP source address spoofing*”. RFC 2827, May 2000.

[11] Nmap. http://www.insecure.org/nmap/.

[12] R. Droms, “*Dynamic host configuration protocol*” RFC 2131, March 1997.

[13] CERT Advisory CA-1998-01, “*Smurf IP denial-of-service attacks*”. http://www.cert.org/advisories/CA-1998-01.html

[14] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, E. Lear, “*Address allocation for private internets*”. RFC 1918, February 1996.

[15] CERT Advisory CA-1996-26, “*Denial-of-service attack via ping*”. http://www.cert.org/advisories/CA-1996-26.html

[16] O. Arkin, “*ICMP Usage in Scanning: The Complete Know How*”. http://www.sys-security.com/archive/papers/ICMP_Scanning_v3.0.pdf

[17] N. Freed, “*Behaviour of and requirements for Internet firewalls*”. RFC 2979, October 2000

October 2000.

[18] R. Braden, “*Requirements for Internet hosts: communication layers*”. RFC 1122, October 1989.

[19] D. B. Chapman, E. D. Zwicky, “*Building Internet Firewalls*”. O’Reilly & Associates, Inc., 1995.

[20] M. Grennan, “*Firewall and proxy server HOWTO*”. Version 0.80, February 2000. http://www.tldp.org/HOWTO/Firewall-HOWTO.html

[21] T. Bowden, B. Bauer, J. Nerin, “*The /proc filesystem*”. November 2000. http://hr.uoregon.edu/davidrl/Documentation/filesystems/proc.txt

[22] IP Tables. http://www.iptables.org/