

Network Security Using Firewalls

- Problems related to security and access can come from outside or inside. These can be malicious in intent, the result of "just looking around", or accidental.
- It is important to verify the cause and intent of the intrusion by careful examination of log files.
- Once the intrusion has been characterized, steps can be taken to block the appropriate addresses and ports using a firewall.
- We will discuss the various techniques for controlling access to a network and protecting critical components using firewalls.
- The following are some of the more important methods of attack and ways of protecting yourself against them:

Unauthorized access

- Unauthorized users outside your company attempt to connect to company file servers. Accessing an NFS server for example.
- These attacks can be avoided by carefully specifying who can gain access through services such as NFS and SAMBA

Exploitation of known weaknesses in applications

- Some applications and network services were not originally designed with strong security features and are inherently vulnerable to attack. Examples are remote access services such as rlogin, rexec, etc.
- The best way to protect against this type of attack is to disable any vulnerable services or severely restrict them.

Denial of service

- Denial of service attacks are designed to overload a service and cause it to cease functioning or prevent others from accessing the service or program.
- These may be performed at the network layer by sending carefully crafted and malicious packets that cause network connections to fail.
- They may also be performed at the application layer, where carefully crafted application commands are given to a program that cause it to become extremely busy or stop functioning.
- Preventing network traffic from suspicious sources from reaching your hosts and preventing suspicious program commands and requests are the best ways of minimizing the risk of a denial of service attack.

IP Spoofing

- IP Spoofing is a security exploit where an Intruder attempts to send packets to a system which appear to originate from a source other than the Intruder's own.
- If the target system already has an authenticated TCP session with another system on the same IP network, and it mistakenly accepts a spoofed IP packet, then it may be induced to execute commands in that packet, as though they came from the authenticated connection.
- The attacker pretends to be an innocent host by following IP addresses in network packets. For example, a well-documented exploit of the BSD **rlogin** service can use this method to mimic a TCP connection from another host by guessing TCP sequence numbers.
- Verifying the authenticity of packets and commands will prevent this attack. Prevent routing with invalid source addresses.
- Introduce unpredictability into connection control mechanisms, such as TCP sequence numbers and the allocation of dynamic port addresses.

Eavesdropping

- A host is configured to "listen" to and capture data that is flowing on the network. Usernames and passwords are thus captured from user login network connections. Broadcast networks like Ethernet are especially vulnerable to this type of attack.
 - One of the best ways to protect against this attack is to use of data encryption and secure connections (secure shell).

IP Firewalling

- Firewalls are very effective in preventing or reducing unauthorized access, network layer denial of service, and IP spoofing attacks.
- They are not very useful in avoiding exploitation of weaknesses in network services or programs and eavesdropping. These can be mitigated using other techniques.
- A firewall is a secure and trusted machine running specialized software that is inserted between a private network and a public network. The firewall software is configured with a set of rules that determine which network traffic will be allowed to pass and which will be blocked.
- The most sophisticated firewall arrangement involves a number of separate machines and is known as a **perimeter network**.
- Two machines act as "**filters**" called chokes to allow only certain types of network traffic to pass, and between these chokes reside network servers such as a mail gateway or a World Wide Web proxy server.

- This configuration can be very safe and easily allows quite a great range of control over who can connect both from the inside to the outside, and from the outside to the inside.
- Typically though, firewalls are single machines that serve all of these functions. These are a little less secure, but cheaper and easier to manage than the more sophisticated perimeter networks.
- The Linux kernel provides a range of built-in features that allow it to function quite effectively as an IP firewall.
- The network implementation includes code to do IP filtering in a number of different ways, and provides a mechanism to dynamically design and implement the sort of rules that the firewall will apply to packets entering and leaving the network.

IP Filtering

- This is simply a mechanism that decides which types of IP packets will be processed normally and which will be deleted and completely ignored, as if they had never been received.
- We can apply different criteria to determine which packets are to be filtered:
 - Protocol type: TCP, UDP, ICMP, etc.
 - Socket number (for TCP/UPD)
 - Datagram type: SYN/ACK, data, ICMP Echo Request, etc.
 - Datagram source address.
 - Datagram destination address.
- It is important to understand that IP filtering is a network layer facility. This means it doesn't understand anything about the application using the network connections, only about the connections themselves.
- For example, you may deny users access to your internal network on the default telnet port, but relying on IP filtering alone will not stop them from using the telnet program with a port that you do allow to pass through your firewall.
- Using proxy servers for each service that you allow across your firewall can prevent this. The proxy servers understand the application they were designed to proxy and can therefore prevent abuses, such as using the telnet program to get past a firewall by using the World Wide Web port.
- The IP filtering ruleset is made up of many combinations of the criteria listed previously.
- For example, say you wanted to allow web users within your network to have no access to the Internet except to use other sites' web servers.

- You would configure your firewall to allow forwarding of:
- Packets with a source address on your network, a destination address of anywhere, and with a destination port of 80.
- Packets with a destination address of your network and a source port of 80 from a source address of anywhere
- We have used two rules here. We have allowed our data to go out, but also the corresponding reply data to come back in. Linux simplifies this and allows us to specify both rules in one command.

Setting Up Linux for Firewalling

- To build a Linux IP firewall, it is necessary to have a kernel built with IP firewall support and the appropriate configuration utility.
- In all production kernels prior to the 2.2 series, you would use the *ipfwadm* utility.
- The 2.2.x kernels marked the release of the third generation of IP firewall for Linux called ***IP Chains***. IP chains use a program similar to *ipfwadm* called *ipchains*.
- Linux kernels 2.3.15 and later support the fourth generation of Linux IP firewall called ***netfilter***. The ***netfilter*** code is the result of a major redesign of the packet handling flow in Linux.
- The ***netfilter*** is a multifaceted utility, providing direct backward-compatible support for both *ipfwadm* and *ipchains* as well as a new configuration utility called *iptables*.

Netfilter and iptables

- The old IP chains significantly improved the efficiency and management of firewall rules but that utility still has some drawbacks.
- The way it processes datagrams is still complex, especially in conjunction with firewall-related features like **IP masquerade** and **Network Address Translation (NAT)**.
- Part of this complexity existed because IP masquerade and NAT were developed independently of the IP firewalling code and integrated later, rather than having been designed as a true part of the firewall code from the start.
- The other problem is how the individual chains work. In particular, the "**input**" chain described input to the IP networking layer as a whole. The input chain affected both datagrams to be **destined for** this host and datagrams to be **routed by** this host.
- This was somewhat counterintuitive because it confused the function of the input chain with that of the forward chain, which applied only to datagrams to be forwarded, but which always followed the input chain.
- If you wanted to treat datagrams for this host differently from datagrams to be forwarded, it was necessary to build complex rules that excluded one or the other. The same problem applied to the output chain.
- The **Netfilter** implementation addresses both the complexity and the rigidity of older solutions by implementing a generic framework in the kernel that streamlines the way datagrams are processed and provides a capability to extend filtering policy without having to modify the kernel.
- Netfilter is composed of two parts: the kernel portion that is known as **netfilter**, and an extensible configuration tool called **iptables** has been created.
- The three main areas of improvement over the older ipchains are as follows:
 - Changes to how the built-in chains process packets
 - The separation of NAT from filtering rules
 - Provision of powerful extensions, including the "state" extension.
- In IP chains, the input chain applies to all datagrams received by the host, irrespective of whether they are destined for the local host or routed to some other host.
- In **netfilter**, the **input** chain applies only to datagrams destined for the **local host**, and the **forward** chain applies only to datagrams destined for a **remote** host.
- Similarly, in IP chains, the output chain applies to all datagrams leaving the local host, irrespective of whether the datagram is generated on the local host or routed from some other host.

- In *netfilter*, the **output** chain applies only to datagrams generated on the **local host** and does not apply to datagrams being routed (forwarded) from another host.
- In *ipchains*, a packet passes through the packet filtering code as shown in Figure 1.

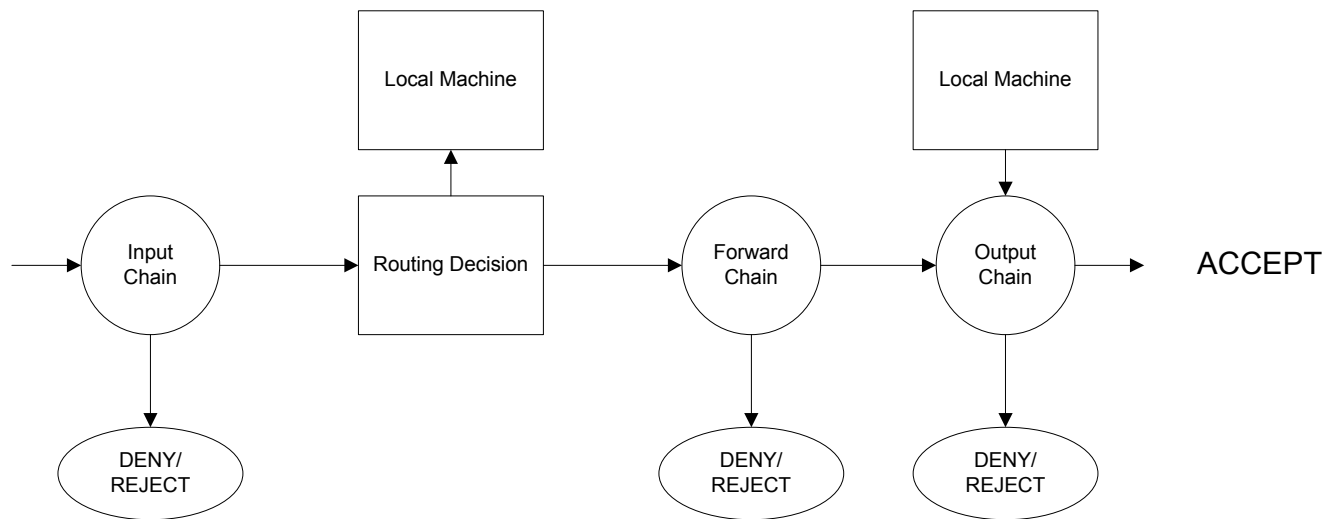


Figure 1: IP Chains

- The circles represent the packet filter checks. First the packet gets checked against the "**input**" chain. If it passes, it goes on to be routed: this decides if it's a local packet, or if it should pass through the box.
- If it's a local packet, it's delivered to the target machine. Otherwise, it's checked against the "**forward**" chain. If that checks out, it passes through the "**output**" chain before being sent on.
- Similarly, if a program sends a packet out, it goes through the "**output**" chain before leaving.
- Figure 2 illustrates how *iptables* does this. The input and output chains have moved: the input chain is only examined if the packet is really destined for us, not for packets that are merely passing through.

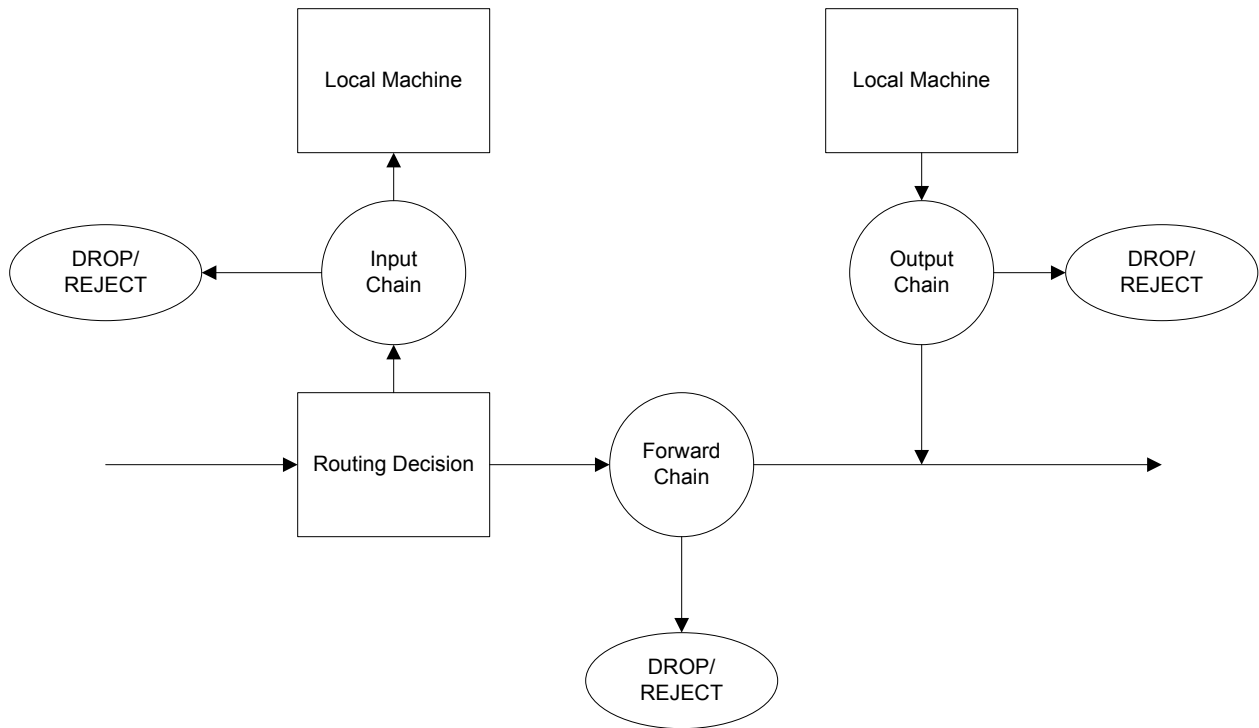


Figure 2: IP Tables

- The output chain is only used for locally-generated packets. This change means that every packet passes through one, and only one chain.
- Besides being more efficient, it also turns out to be much simpler to use. If you want to control packets passing through your box, all you have to do is add more checks in the "**forward**" chain.
- If you want to control packets coming into your box itself, you use the input chain. To emphasize this change, the chain names in *iptables* are upper case, i.e. **INPUT**, **FORWARD** and **OUTPUT**.
- The netfilter implementation with *iptables* simplifies firewall configurations significantly. For a service to be routed across the firewall host, but not terminate on the local host, only two rules are required: one each for the forward and the reverse directions in the forward chain.
- The Linux *netfilter* implementation has the ability to emulate the *ipfwadm* and *ipchains* interfaces. Emulation makes transition to the new generation of firewall software a little easier.
- The two *netfilter* kernel modules called *ipfwadm.o* and *ipchains.o* provide backward compatibility for *ipfwadm* and *ipchains*.

- Netfilter adds an additional layer of abstraction by having tables that incorporate the chains. There are three tables within netfilter: **filter** (default table), **nat** (Network Address Translation), and **mangle**.
- Each table above contains its own set of predefined chains. The filter table is the default table. The other tables are specified by a command line option.

The filter table

- This table provides features such as:
 - Chain-related operations on the three built-in chains (INPUT, OUTPUT, and FORWARD) and on user-defined chains.
 - Target disposition (ACCEPT or DROP)
 - IP header field match operations for protocol, source and destination address, input and output interfaces, and fragment handling
 - Match operations on the TCP, UDP, and ICMP headers
- The filter table has two kinds of feature extension modules, **target** extensions and **match** extensions.
- The target extensions include the **REJECT** packet disposition and the **LOG** functionality.
- The match extensions include modules to support matching on the following:
 - The current connection state
 - Port lists (supported by the multipart module)
 - The hardware Ethernet MAC source address
 - The packet sender's user, group, process, or process group ID
 - The IP header Type of Service (TOS) field (possibly set by the mangle table module)
 - The iptables mark field (set by the mangle table module)
 - Rate-limited packet matching

- Packet-filtering features in **iptables** includes the following:
 - Connection-state maintenance for TCP, UDP, and ICMP exchanges
 - Access to the IP headers TOS field
 - Access to the MAC source address
 - Extended logging capabilities
 - Rate-limited packet matching
 - Extended packet-rejection features
 - Queuing packets to user-space programs
 - Filtering outgoing packets by user ID, group ID, or process group ID
 - Source and destination port lists
 - Access to the TCP state flags
 - Access to the TCP options field
- Source and destination ports can contain a comma-separated list of up to 15 ports (However, port lists and ranges cannot be combined).
- All TCP state flags can be inspected, and filtering decisions can be made based on the results.
- TCP connection state and ongoing UDP exchange information can be maintained, allowing packet filtering on an ongoing basis rather than on a stateless, packet-by-packet basis.
- Accepting packets that are part of an established connection allows bypassing the overhead of checking the rule list for each packet. When the initial connection is accepted, subsequent packets can be recognized and allowed.
- Traditionally the TOS field was unused but with the newer Differentiated Services definitions by intermediate routers, it is used for local packet prioritizing, routing and forwarding among local hosts and the local router.
- Incoming packets can be filtered by the MAC source address.
- Individual filter log messages can be prefixed with user-defined strings. Messages can be assigned kernel logging levels as defined for **/etc/syslog.conf**. This allows logging to be turned on and off, and for the log output files to be defined, in **/etc/syslog.conf**.
- Packet matches can be limited to an initial burst rate, after which a limit is imposed by the number of allowed matches per second.

- If match limiting is enabled, the default is that, after an initial burst of five matched packets, a rate limit of three matches per hour is imposed.
- In other words, if the system were flooded with ping packets, for example, the first five pings would match.
- After that, a single ping packet could be matched 20 minutes later, and another one could be matched 20 minutes after that, regardless of how many echo - requests were received.
- This provides a measure of defense against denial of service attacks.
- The REJECT target can optionally specify which ICMP (or RST for TCP) error message to return.
- The IPv4 standard requires TCP to accept either RST or ICMP as an error indication, although RST is the default TCP behavior. iptable's default is to return nothing (DROP) or else to return an ICMP error (REJECT).
- Along with REJECT, another special-purpose target is QUEUE. Its purpose is to hand off the packet to a user-space program for handling. If there is no waiting program, the packet is dropped.
- RETURN is another special-purpose target. Its purpose is to return from a user-defined chain before rule matching on that chain has completed.
- Locally generated outgoing packets can be filtered based on the user, group, process, or process group ID of the program generating the packet.
- Thus, access to remote services can be authorized at the packet-filtering level on a per-user basis.
- Note that this is a specialized option for multiuser, multipurpose hosts because firewall routers shouldn't have user accounts.

The mangle Table

- Supports two target extensions: **MARK** and **TOS**.
- The **MARK** module supports assigning a value to the packet's **mark** field that iptables maintains.
- The **TOS** module supports setting the value of the **TOS** field in the IP header.
- The mangle table allows **marking**, or associating a Netfilter-maintained value, with the packet, as well as changing the packet's **TOS** field for specialized purposes before making a local routing/forwarding decision.
- The mangle table has two built-in chains:
 - The **PREROUTING** chain specifies changes to incoming packets as they arrive at an interface, before any routing or local delivery decision has been made.
 - The **OUTPUT** chain specifies changes to locally generated outgoing packets.
- In terms of the **TOS** field, the local Linux router can be configured to honor the **TOS** flags set by the mangle table or as set by the local hosts.

The nat Table

- The **nat** table has target extension modules for source and destination **address translation** and for **port translation**.
- These modules support the following forms of **NAT (Network Address Translation)**:
 - **SNAT** - Source NAT
 - **DNAT** - Destination NAT
- **MASQUERADE** - A specialized form of source NAT for connections that are assigned a temporary, changeable, dynamically assigned IP address (such as a phone dial-up connection)
- **REDIRECT** - A specialized form of destination NAT that redirects the packet to the local host, regardless of the address in the IP header's destination field

- Traditional, **unidirectional outbound** NAT is used for networks using private addresses.
 - Basic NAT provides address translation only and is used to map local private source addresses to one of a block of public addresses.
 - NAPT (Network Address Port Translation) is used to map local private source addresses to a single public address (for example, Linux masquerading).
- **Bidirectional** NAT provides two-way address translation for both outbound and inbound connections.
 - A use of this is bidirectional address mapping between IPv4 and IPv6 address spaces.
- **Twice** NAT
 - Two-way source and destination address translation allows both outbound and inbound connections.
 - Twice NAT can be used when the source and destination networks' address spaces collide as a result of one site mistakenly using public addresses assigned to someone else.
 - Twice NAT also can be used as a convenience when a site was renumbered or assigned to a new public address block and the site administrator didn't want to administer the new address assignments locally at that time.
- The NAT table allows for modifying a packets source address or destination address and port using three built-in chains.
- The **PREROUTING** chain
 - Specifies destination changes to incoming packets before passing the packet to the routing function (DNAT).
 - Changes to the destination address can be to the local host (transparent proxying, port redirection) or to a different host for host forwarding (**ipmasqadm** functionality, port forwarding in Linux parlance) or load sharing.
- The **OUTPUT** chain
 - Specifies destination changes to locally generated outgoing packets before the routing decision has been made (DNAT, REDIRECT).
 - This is usually done to transparently redirect an outgoing packet to a local proxy, but it can also be used to port-forward to a different host.

- The **POSTROUTING** chain
 - Specifies source changes to outgoing packets being routed through the box (SNAT, MASQUERADE).
 - The changes are applied after the routing decision has been made.

Using iptables

- The **iptables** utility is used to configure **netfilter** filtering rules. Its syntax is very similar to the **ipchains** command, but differs in one very significant respect: it is **extensible**.
- Extensibility here refers to the fact that its functionality can be extended without recompiling it. It does this by using shared libraries.
- The general syntax of the **iptables** command is:

iptables command rule-specification extensions

Commands

- There are a number of ways we can manipulate rules and rulesets with the **iptables** command. Those relevant to **IP firewalling** are:

-A chain

- Append one or more rules to the end of the nominated chain. If a hostname is supplied as either a source or destination and it resolves to more than one IP address, a rule will be added for each address.

-I chain rulenum

- Insert one or more rules to the start of the nominated chain. Again, if a hostname is supplied in the rule specification, a rule will be added for each of the addresses to which it resolves.

-D chain

Delete one or more rules from the specified chain matching the rule specification.

-D chain rulenum

Delete the rule residing at position ***rulenum*** in the specified chain. Rule positions start at 1 for the first rule in the chain.

-R chain rulenum

Replace the rule residing at position ***rulenum*** in the specific chain with the supplied rule specification.

-L [chain]

List the rules of the specified chain, or for all chains if no chain is specified.

-F [chain]

Flush the rules of the specified chain, or for all chains if no chain is specified.

-Z [chain]

Zero the datagram and byte counters for all rules of the specified chain, or for all chains if no chain is specified.

-N chain

Create a new chain with the specified name. A chain of the same name must not already exist. This is how user-defined chains are created.

-X [chain]

Delete the specified user-defined chain, or all user-defined chains if no chain is specified.

-P chain policy

Set the default policy of the specified chain to the specified policy. Valid firewalling policies are **ACCEPT**, **DROP**, **QUEUE**, and **RETURN**.

- **ACCEPT** allows the datagram to pass.
- **DROP** causes the datagram to be discarded.
- **QUEUE** causes the datagram to be passed to userspace for further processing.
- **RETURN** target causes the IP firewall code to return to the Firewall Chain that called the one containing this rule, and continue starting at the rule after the calling rule.

Rule specification parameters

- There are a number of *iptables* parameters that constitute a rule specification. Wherever a rule specification is required, each of these parameters must be supplied or their default will be assumed.

-p [!]*protocol*

- Specifies the protocol of the datagram that will match this rule. Valid protocol names are tcp, udp, icmp, or an IP protocol number (*/etc/protocols*).

If the ! character is supplied, the rule is negated and the datagram will match any protocol other than the specified protocol. If this parameter isn't supplied, it will default to match all protocols.

-s [!]*address[/mask]*

Specifies the source address of the datagram that will match this rule. The address may be supplied as a hostname, a network name, or an IP address.

-d [*address[/mask]*

Specifies the destination address and port of the datagram that will match this rule. The coding of this parameter is the same as that of the -s parameter.

-j *target*

Specifies what action to take ("jump to") when this rule matches. Valid targets are ACCEPT, DROP, QUEUE, and RETURN.

You may also specify the name of a user-defined chain where processing will continue.

-i [!] *interface-name*

Specifies the interface on which the datagram was received. Again, the inverts the result of the match. If the interface name ends with "+" then any interface that begins with the supplied string will match. For example, -i ppp+ would match any PPP network device and -i ! eth+ would match all interfaces except Ethernet devices.

-i [!]interface-name

Specifies the interface on which the datagram is to be transmitted. This argument has the same coding as the *-i* argument.

[!] -f

- Specifies that this rule applies only to the second and later fragments of a fragmented datagram, not to the first fragment.

Options

The following *iptables* options are more general in nature. Some of them control the more esoteric features of the *netfilter* software:

-v

- Causes *iptables* to be verbose in its output; it will supply more information.

-n

- Causes *iptables* to display IP address and ports as numbers without attempting to resolve them to their corresponding names.

-x

- Causes any numbers in the *iptables* output to be expanded to their exact values with no rounding.

- line-numbers

- causes line numbers to be displayed when listing rulesets. The line number will correspond to the rule's position within the chain.

Extensions

- The **iptables** utility is extensible through optional shared library modules. There are some standard extensions that provide some of the features **ipchains** provided.
- To make use of an extension, you must specify its name through the **-m name** argument to **iptables**.
- The following list shows the **-m** and **-p** options that set up the extension's context, and the options provided by that extension.

TCP Extensions

These are used with: **-m tcp -p tcp**

--sport [!] [port[:port]]

- Specifies the source port that the datagram must be using to match this rule.
- Ports are specified as a range by specifying the upper and lower limits of the range using the colon as a delimiter.
- For example, **20: 25** described all of the ports numbered 20 up to and including 25. The **!** character may be used to negate the values.

--dport [!] [port[:port]]

- Specifies the destination port that the datagram must be using to match this rule. The argument is coded identically to the **-sport** option.

--tcp-flags [!] mask comp

- Specifies that this rule should match when the TCP flags in the datagram match those specified by **mask** and **comp**. **mask** is a comma-separated list of flags that should be examined when making the test.
- **comp** is a comma-separated list of flags that must be set for the rule to match. Valid flags are: **SYN**, **ACK**, **FIN**, **RST**, **URG**, **PSH**, **ALL** or **NONE**.

[!] --syn

- Specifies the rule to match only datagrams with the **SYN** bit set and the **ACK** and **FIN** bits cleared.
- Datagrams with these options are used to open TCP connections, and this option can therefore be used to manage connection requests.
- This option is shorthand for:

--tcp-flags SYN,RST,ACK SYN

- When you use the negation operator, the rule will match all datagrams that do not have both the **SYN** and **ACK** bits set.

UDP Extensions

These are used with: ***-m udp -p udp***

--sport [!][port[:port]]

- Specifies the port that the datagram source must be using to match this rule.
- Ports may be specified as a range, by specifying the upper and lower limits of the range as described earlier.

--dport [!] [port[:port]]

Specifies the port that the datagram destination must be using to match this rule.
The argument is coded identically to the ***--sport*** option.

ICMP Extensions

- These are used with: ***-m icmp -p icmp***

--icmp-type [!] typename

- Specifies the ICMP message type that this rule will match.
- The type may be specified by number or name.
- Some valid names are: **echo-request**, **echo-reply**, **source-quench**, **time-exceeded**, **destination-unreachable**, **network-unreachable**, **host-unreachable**, **protocol-unreachable** and **port-unreachable**.

MAC Extensions

- These are used with: ***-m mac***
--mac-source [!] address
- Specifies the host's Ethernet address that transmitted the datagram that this rule will match.
- This only makes sense in a rule in the input or forward chains because we will be transmitting any datagram that passes the output chain.

Examples

- Here is an example that drops malformed packets. Malformed packets (TCP, UDP and ICMP packets too short for the firewalling code to read the ports or ICMP code and type) are dropped when such examinations are attempted.

```
# iptables -A INPUT -m limit --limit 1/minute --limit-burst 5 -m unclean -j LOG \
--log-prefix "[firewall] unclean packet: "
# iptables -A INPUT -m unclean -j DROP
```

- Note that the **unclean** option is considered experimental at this stage.
- The following rule will drop any fragments going to 192.168.1.1:

```
# iptables -A OUTPUT -f -d 192.168.1.1 -j DROP
```

The following rule will drop all incoming packets from the 192.168.1.0 subnet:

```
# iptables -A INPUT -s 192.168.1.0/24 -j DROP
```

- The following rule specifies that all flags should be examined ("ALL" is synonymous with "SYN,ACK,FIN,RST,URG,PSH"), but only SYN and ACK should be set. There is also an argument "NONE" meaning no flags.

```
# iptables -A INPUT --protocol tcp --tcp-flags ALL SYN,ACK -j DROP
```

- It is sometimes useful to allow TCP connections in one direction, but not the other. For example, you might want to allow connections to an external WWW server, but not connections from that server.
- The obvious approach would be to block TCP packets coming from the server. Unfortunately, TCP connections require packets going in both directions to work at all.

- The solution is to block only the packets used to request a connection. These packets are called **SYN** packets (packets with the SYN flag set, and the FIN and ACK flags cleared).
- By disallowing only these packets, we can stop attempted connections.
- The "--syn" flag is used for this: it is only valid for rules which specify TCP as their protocol.
- For example, to stop TCP connection attempts from 192.168.1.1:

```
# iptables -A INPUT -p tcp -s 192.168.1.1 --syn -j DROP
```

- This flag can be inverted by preceding it with a "!", which means every packet other than the connection initiation.

Simple Firewall Example

- Suppose that we are using a Linux-based firewall machine to allow local users to access servers on the Internet, but to allow no other traffic to be passed.
- If the network has a **24-bit network mask** (class B) and has an address of **142.232.66.0**, we'd use the following *iptables* rules:

```
# iptables -F FORWARD
# iptables -P FORWARD DROP
# iptables -A FORWARD -m tcp -p tcp -s 0/0 --sport 80 -d 142.232.66.0/24 --syn \
-j DROP
# iptables -A FORWARD -m tcp -p tcp -s 142.232.66.0/24 --dport 80 -d 0/0 \
-j ACCEPT
# iptables -A FORWARD -m tcp -p tcp -d 142.232.66.0/24 --sport 80 -s 0/0 -j ACCEPT
```

- Note that *iptables* doesn't support the **-b** (bidirectional) option, so we must supply a rule for each direction.

multiport filter Table Match Extension

- **multiport** port lists can include up to 15 ports per list.
- The following rule blocks incoming packets destined for the UDP ports associated with NFS and lockd:

```
# iptables -A INPUT -i eth0 -p udp -m multiport \  
--destination-port 2049,4045 -j DROP
```

- The following rule blocks outbound connection requests to high ports associated with the TCP services NFS, socks, and squid:

```
# iptables -A OUTPUT -o eth0 -p tcp -m multiport \  
--destination-port 2049,1080,3128 -j DROP
```

Logging

- Netfilter improves on previous versions by providing the ability to restrict the rate of matches, such as for suppressing extensive log messages, for example when a host makes a number of connections in a predefined interval.
- This can help to prevent some types of denial of service attacks. The detailed logging that netfilter provides enables a firewall administrator to not only troubleshoot potential system problems, but also to track a potential intrusion, and correlate it with other systems and events.
- The example below shows how to reduce the number of packets originating from the 192.168.1.1 host that are logged:

```
# iptables -A INPUT -s 192.168.1.1 -m limit --limit 1/second -j LOG
```

Rate Limiting and DoS Attacks

- Iptables provides an option for limiting the rate of packets handled on an interface over a given period of time.
- This feature is very useful in preventing Denial of Service (DoS) attacks.
- Consider the following example that provides SYN flood protection.

```
# iptables -A INPUT -p TCP --syn -m limit --limit 5/second -j ACCEPT
```

- The above rule matches packets with the SYN bit set and the ACK and FIN bits cleared.
- What this means is it affects only packets wishing to open a connection, not packets involved in previously created connections.
- This is specified with the **--syn** command appended to the **-p TCP** option.
- Rate limiting is then specified with **-m limit** and the actual rate to accept with **--limit 5/second**.
- Rate-limited matching is also useful for reducing the number of log messages that would be generated during a flood of logged packets.
- The following two rules, in combination, will limit acceptance of incoming ping messages to one per second when an initial five echo-requests are received within a second:

```
# iptables -A INPUT -i eth0 \  
-p icmp -icmp-type echo-request \  
-m limit --limit 1/second -j ACCEPT
```

```
# iptables -A INPUT -i eth0 \  
-p icmp -icmp-type echo-request -j DROP
```

state Filter Table Match Extensions

- Stateless filters examine traffic on a packet-by-packet basis alone. Each packet's particular combination of source and destination addresses and ports, the transport protocol, and the current TCP state flag combination is examined without reference to any previous traffic or context.
- The state extension module provides additional monitoring and recording technology to augment the stateless, static packet-filter technology.
- State information is recorded when a TCP connection or UDP exchange is initiated. Subsequent packets are examined based not only on the static tuple information, but also within the context of the ongoing exchange.
- In other words, some of the contextual knowledge usually associated with the upper TCP transport layer, or the UDP application layer, is brought down to the filter layer.
- After the exchange is initiated and accepted, subsequent packets are identified as part of the established exchange.
- In terms of packet flow performance and firewall complexity, the advantages are more obvious in terms of TCP. Flows are primarily a firewall performance and optimization technology.
- The main goal of flows is to allow bypassing the firewall inspection path for a packet.
- Much faster TCP packet handling is obtained in some cases because the remaining firewall filters can be skipped if the TCP packet is immediately recognized as part of an established connection.
- The main disadvantage is that maintaining a state table requires more memory than standard firewall rules alone.
- Routers with thousands of simultaneous connections, for example, would require enormous amounts of memory to maintain state table entries for each connection.
- State maintenance is often done in hardware for performance reasons, where associative table lookups can be done simultaneously or in parallel.
- Whether implemented in hardware or software, state engines must be capable of reverting a packet to the traditional path if memory isn't available for the state table entry.
- In addition, table creation, lookup, and teardown take time in software. The additional processing overhead is a loss in many cases.
- State maintenance is a win for ongoing exchanges such as an FTP transfer or a UDP streaming multimedia session.
- Both types of data flow represent potentially large numbers of packets (and filter rule match tests). State maintenance is not a firewall performance win for a simple DNS or NTP client/server exchange, however.

- State buildup and teardown can easily require as much processing and more memory than simply traversing the filter rules for these packets.
- TCP connection state and ongoing UDP exchange information can be maintained allowing network exchanges to be filtered as **NEW**, **ESTABLISHED**, or **RELATED**:
- **NEW** is equivalent to the initial TCP **SYN** request, or to the **first UDP packet**.
- **ESTABLISHED** refers to the ongoing TCP **ACK** messages after the connection is initiated, to subsequent UDP **datagrams** exchanged between the **same hosts** and **ports**, and to ICMP **echo -reply** messages sent in response to a previous **echo** request.
- **RELATED** currently refers only to ICMP error messages. **FTP** secondary connections are managed by the additional **FTP** connection tracking support module.
- With the addition of that module, the meaning of **RELATED** is extended to include the secondary **FTP** connection.
- Let us assume that we have a Linux machine with two network cards that is being used as a firewall. The following rules can be used to forward packets between the **private network** and the **Internet**.
- First we forward all packets from **eth1** (internal network) to **eth0** (the internet).

```
#iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT
```

- Forward packets that are part of existing and related connections from eth0 to eth1.

```
#iptables -A FORWARD -i eth0 -o eth1 \  
-m state --state ESTABLISHED,RELATED -j ACCEPT
```

- Allow packets in to firewall itself that are part of existing and related connections.

```
#iptables -A INPUT -i eth0 \  
-m state --state ESTABLISHED,RELATED -j ACCEPT
```

- Note, in the above two rules, a connection becomes **ESTABLISHED** in the iptables **PREROUTING** chain upon receipt of a **SYN/ACK** packet that is a response to a previously sent **SYN** packet.
- The **SYN/ACK** packet itself is considered to be part of the established connection, so no special rule is needed to allow the **SYN/ACK** packet itself.
- The following is a rule pair for a local DNS server operating as a cache-and-forward name server.
- A DNS forwarding name server uses server-to-server communication. DNS traffic is exchanged between **source** and **destination** ports **53** on both hosts.

- The UDP client/server relationship can be made explicit. The following rules explicitly allow outgoing (NEW) requests, incoming (ESTABLISHED) responses, and any (RELATED) ICMP error messages:

```
#iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
```

```
#iptables -A OUTPUT --out-interface <interface> -p udp \  
    -s $IPADDR --source-port 53 -d $NAME-SERVER --destination-port 53 \  
    -m state --state NEW,RELATED -j ACCEPT
```

- DNS uses a simple query-and-response protocol. What about an application that can maintain an ongoing connection for extended periods of time, such as an FTP control session or a SSH session?
- If the state table entry is cleared out prematurely for some reason, future packets will not have a state entry to be watched against to be identified as part of an ESTABLISHED exchange.
- The following rules for a **ssh** connection allow for that possibility:

```
#iptables -A INPUT -m state \  
    --state ESTABLISHED,RELATED -j ACCEPT
```

```
#iptables -A OUTPUT -m state \  
    --state ESTABLISHED,RELATED -j ACCEPT
```

```
#iptables -A OUTPUT --out-interface <interface> -p tcp \  
    -s $IPADDR --source-port $UNPRIVPORTS \  
    -d $REMOTE_SSH_SERVER --destination-port 22 \  
    -m state --state NEW -j ACCEPT
```

```
#iptables -A OUTPUT --out-interface <interface> -p tcp ! -syn \  
    -s $IPADDR --source-port $UNPRIVPORTS  
    -d $REMOTE_SSH_SERVER --destination-port 22 \  
    -j ACCEPT
```

```
#iptables -A INPUT --in-interface <interface> -p tcp -syn \  
    -s $REMOTE_SSH_SERVER --source-port 22 \  
    -d $IPADDR --destination-port $UNPRIVPORTS \  
    -j ACCEPT
```

NAT Table Target Extensions

- As mentioned earlier, iptables supports four target types of NAT:
 - Source NAT (SNAT)
 - Destination NAT (DNAT)
 - Masquerading (MASQUERADE)
 - REDIRECT
- These targets only valid in the NAT table. **SNAT** and **MASQUERADING** is done in the **POSTROUTING** chain.
- **DNAT** and **REDIRECT** is done in the **PREROUTING** chain.
- NAT is the process of altering the packet headers as they move outbound through a gateway and altering the packet header replies as they come back through the gateway.
- Source NAT (SNAT) refers to changing the Source IP address of the packets, i.e., where they are coming from.
- Destination NAT (DNAT) refers to changing the Destination IP address of the packets, i.e., where they are going to.
- We can insert NAT rules at two places:
 - Destination NAT can be done at the PREROUTING point, which is as the packet comes in.
 - Source NAT can be done at the POSTROUTING point, just before it goes out.
- Consider the case where we have a single static IP address but we wish to have all the systems on a home network to use that IP address to connect to the Internet.
- We can use SNAT to map **private IP** addresses of interfaces on the internal LAN to a public **static IP** address.
- SNAT performs this mapping when a client running on one of the internal machines initiates a TCP connection (SYN) through eth0.
- The following example is one way to accomplish this:

```
#iptables -A POSTROUTING -t nat \  
-s 192.168.0.0/24 -o eth0 \  
-j SNAT --to-source 1.2.3.4
```
- The above will append a rule to the POSTROUTING chain that applies to packets going out the external interface (eth0).

- It will jump to the Source NAT target and change the private source address to the external IP address 1.2.3.4.
- **DNAT** modifies the destination IP address of all incoming packets using a specified address or address range, including a port or port range.
- This technique can be used to support of multiple Servers ("port forwarding") and Load Sharing.
- We can use NAT to change the destination address of an incoming packet. Typically this is done to pass an inbound connection to an internal server that is not visible to the Internet.
- Usually such services are hosted on a private network. For packets arriving on that service's port, the firewall changes the destination address to that of the local server's network interface, and forwards the packet to the private machine.
- The reverse is done for the server's responses to the client. The firewall changes the source address to that of its own external interface and forwards the packet on to the remote client.
- The following example forwards incoming HTTP connections to a local web server:

```
#iptables -t nat -A PREROUTING -i <public interface> -p tcp \  
    --sport 1024:65535 -d <public interface> --dport 80 \  
    -j DNAT --to-destination <local web server address>
```

- DNAT was applied before the packet reached the FORWARD chain so the rule on the FORWARD chain must refer to the internal server's private IP address rather than to the firewall's public address:

```
#iptables -A FORWARD -i <public interface> -o <DMZ interface> -p tcp \  
    --sport 1024:65535 -d <local web server> --dport 80 \  
    -m state --state NEW -j ACCEPT
```

- Because the initial connection request was accepted, a generic FORWARD rule suffices to forward the return traffic from the local server to the Internet:

```
#iptables -A FORWARD -i <DMZ interface> -o <public interface> -p tcp \  
    -m state --state ESTABLISHED,RELATED -j ACCEPT
```

- Note that ongoing traffic from the client must be forwarded as well:

```
#iptables -A FORWARD -i <public interface> -o <DMZ interface> -p tcp \  
    -m state --state ESTABLISHED,RELATED -j ACCEPT
```

TOS Bit Manipulation

- The Type Of Service (TOS) bits are a set of four-bit flags in the IP header. They effect the way packets are treated; each of the four bits has a different purpose and only one of the TOS bits may be set at any time.
- The bit flags are called Type of Service bits because they enable the application transmitting the data to tell the network the type of network service it requires as summarized below.

Minimum delay

- Used when the time it takes for a datagram to travel from the source host to destination host (latency) is most important.
- A service provider might, for example, use both optical fiber and satellite network connections.
- Data carried across satellite connections has farther to travel and their latency is generally therefore higher than for terrestrial-based network connections between the same endpoints.
- Thus a service provider might choose to ensure that datagrams with this type of service set are not carried by satellite.

Maximum throughput

- Used when the volume of data transmitted in any period of time is important.
- There are many types of network applications for which latency is not particularly important but the network throughput is; for example, bulk-file transfers.
- A service provider might choose to route datagrams with this type of service set via high-latency, high-bandwidth routes, such as satellite connections.

Maximum reliability

- Used when it is important that you have some certainty that the data will arrive at the destination without retransmission being required.
- The IP packets may be carried over any number of underlying transmission mediums, some of those not as reliable as others.
- A service provider might make an alternate network available, offering high reliability, to carry IP that would be used if this type of service is selected.

Minimum cost

- Used when it is important to minimize the cost of data transmission.
- Leasing bandwidth on a satellite for a transpacific crossing is generally less costly than leasing space on a fiber-optical cable over the same distance.
- Network providers may choose to provide both and charge differently depending on which you use. In this scenario, the "minimum cost" type of service bit may cause datagrams to be routed via the lower-cost satellite route.

Setting the TOS Bits

I

- The *iptables* tool allows you to specify rules that capture only datagrams with TOS bits matching some predetermined value using the **-m tos** option.
- The TOS bits of IP datagrams can be set for a matching a rule using the **-j TOS** target. TOS bits can be set only on the FORWARD and OUTPUT chains.
- The TOS bits may be specified using mnemonics listed in the table below.

Mnemonic	Hexadecimal
Normal-Service	0x00
Minimize-Cost	0x02
Maximize-Reliability	0x04
Maximize-Throughput	0x08
Minimize-Delay	0x10

- The general syntax used to **match** TOS bits is:

-m tos --tos mnemonic [other-args] -j target

- The general syntax used to **set** TOS bits is:

[other-args] -j TOS --set mnemonic

- Note that these would typically be used together, but they can be used quite independently if a configuration requires it.

- A very common use for setting the TOS bits on a firewall is to clear packets for time-critical (ssh) and high-throughput (ftp) applications faster through the firewall.
- Set **ssh** and **ftp** control connections to "**Minimum Delay**" and **FTP data** to "**Maximum Throughput**".
- This would be done on a **local machine** as follows:

```
# iptables -A OUTPUT -t mangle -p tcp --dport ssh -j TOS \
    --set-tos Minimize-Delay
# iptables -A OUTPUT -t mangle -p tcp --dport ftp -j TOS --set-tos Minimize-Delay
# iptables -A OUTPUT -t mangle -p tcp --dport ftp-data -j TOS \
    --set-tos Maximize-Throughput
```

- On a **stand-alone firewall** this could be accomplished as follows:

```
# iptables -A PREROUTING -t mangle -p tcp --sport ssh \
    -j TOS --set-tos Minimize-Delay
# iptables -A PREROUTING -t mangle -p tcp --sport ftp \
    -j TOS --set-tos Minimize-Delay
# iptables -A PREROUTING -t mangle -p tcp --sport ftp-data \
    -j TOS --set-tos Maximize-Throughput
```

User-defined chains

- The three basic rulesets of the traditional IP firewall code provide us with a mechanism for building firewall configurations that are relatively simple to implement and manage for small networks.
- Large networks often require much more than the small number of firewalling rules we have seen so far. As the number of rules increases, the performance of the firewall deteriorates as more and more tests are conducted on each datagram.
- The other issue is that it is not possible to enable and disable sets of rules atomically. This implies that while you are rebuilding the ruleset your firewall is down, thus exposing your network to attacks.
- The design of IP Firewall chains helps allows the network administrator to create arbitrary sets of firewall rules that we can link to the three **in-built** rulesets.
- We use the **-N** or **--new-chain** options of to create a new chain with any name (30 characters or less). The **-j** option configures the action to take when a datagram matches the rule specification.
- User-defined chains can be deleted using **-X** or **--delete-chain options**.
- The **-j** option specifies that if a datagram matches a rule, further testing should be performed against a **user-defined chain**.
- Consider the following **iptables** commands:

```
# iptables -P INPUT DROP
# iptables -N tcpin
# iptables -A tcpin -p tcp -d 192.168.1.0/24 ssh -j ACCEPT
# iptables -A tcpin -p tcp -d 192.162.1.0/24 www -j ACCEPT
# iptables -A input -p tcp -j tcpin
```

- The first command sets the **default input** chain policy to **DROP**.
- The second command **creates** a user-defined chain called "**tcpin**."
- The next two rules match any datagram that is destined for our local network and either of the **ssh** or **www** ports; datagrams matching these rules are accepted.
- The next rule causes the firewall software to check any datagram of protocol TCP against the **tcpin** user-defined chain.
- They will produce the following chains shown below:

INPUT	Tcpin
-p tcp -j tcpin	-p tcp -d 142.232.0.0/16 ssh -j ACCEPT
	-p tcp -d 142.232.0.0/16 www -j ACCEPT

- The **INPUT** and **tcpin** chains are populated with our rules. Datagram processing always begins at one of the in-built chains.
- Consider what happens when a UDP packet for one of our hosts is received. The packet is received by the INPUT chain and falls through the first rule because it matches only TCP protocols.
- The packet reaches the end of the input chain, meets with the default input chain policy, and is denied.
- To see the user-defined chain in operation consider what happens when we receive a TCP packet destined for the **ssh** port of one of the hosts.
- This time the second rule in the input chain does match and it specifies a target of **tcpin**, our user-defined chain.
- Specifying a user-defined chain as a target causes the packet to be tested against the rules in that chain, so the next rule tested is the first rule in the **tcpin** chain.
- The first rule in our **tcpin** chain does match and specifies a target of **ACCEPT**. We have arrived at target, so no further firewall processing occurs. The packet is accepted.
- Now consider what happens when we reach the end of a user-defined chain. Say we receive a TCP packet destined for a port other than the two we are handling specifically.
- The user-defined chains do not have default policies. When all rules in a user defined chain have been tested, and none have matched, the firewall code acts as though a **RETURN** rule were present.
- In our example, our testing returns to the rule in the INPUT ruleset immediately following the one that moved us to our user-defined chain. Eventually we reach the end of the input chain, which does have a default policy and our datagram is denied.
- User-defined chains are very useful for optimizing large firewall rulesets. They allow the rules to be organized into categories.
- Rather than relying on the straight-through, top-down rule matching that is inherent in the standard chain traversal, packet matching tests can be selectively narrowed down based on specific packet characteristics.
- This also saves time since a packet will no longer be matched against every rule in the firewall ruleset.

Testing a Firewall Configuration

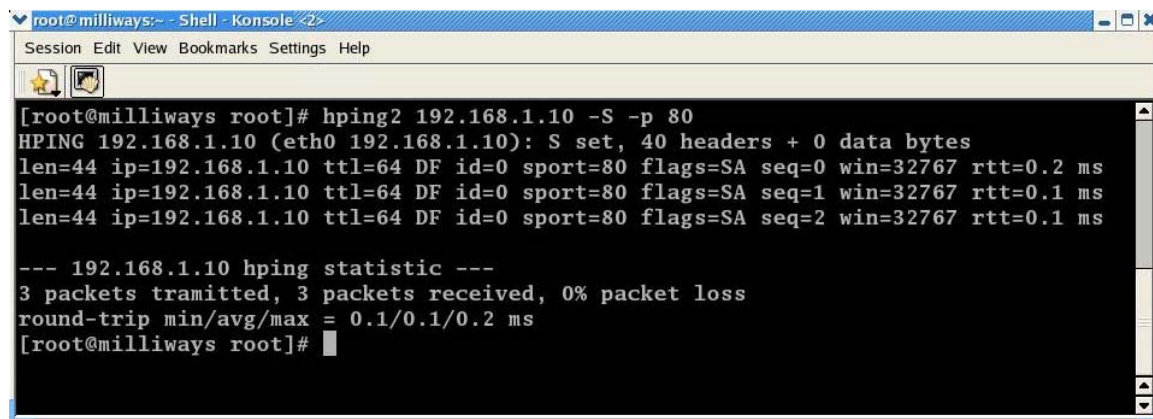
- Before deploying a firewall it is important to validate that it does in fact do what you want the design intended it to do.
- One way to do this is to use a test host outside your network to attempt to penetrate your firewall. This may or may not be feasible depending on the security issues.
- The general test procedure is as follows:
 - Design a series of tests that will determine whether your firewall is actually working as you intend.
 - For these tests you may use any source or destination address, so choose some address combinations that should be accepted and some others that should be dropped.
 - If you're allowing or disallowing only certain ranges of addresses, it is a good idea to test addresses on either side of the boundary of the range-one address just inside the boundary and one address just outside the boundary. This will help verify the specified netmask.
 - If you're filtering by protocol and port number, your tests should also check all the important combinations of these parameters.
 - It is a good idea to write all the rules into a script so you can test and retest easily as you correct mistakes or modifies your design.
 - Execute each test command and note the output.
 - Compare the output of each test against the desired result.
 - If there are any discrepancies, you will need to analyze your ruleset to determine where you've made the error. This is where a script file becomes useful in rerunning the test after correcting any errors in the firewall configuration.
 - It's a good practice to flush your rulesets completely and rebuild them from scratch, rather than to make changes dynamically.
- It is absolutely imperative to design a series of exhaustive tests. While this can sometimes be almost as much work as designing the firewall configuration, it is also the best way of verifying that the firewall design is providing the security expected of it.

Firewall Testing using Reconnaissance Tools (*hping2*)

- Before an attacker can successfully attack or exploit an organization, a certain level of reconnaissance must be performed.
- Enough information must be acquired about the target to have a solid understanding of the network, services, and probable vulnerabilities of the network under attack.
- A very powerful reconnaissance tool is hping2. hping2 is a network tool that sends custom ICMP, UDP, and TCP packets and displays target replies the same way ping does with ICMP replies.
- In addition to the normal ICMP functionality, hping2 can handle fragmentation, arbitrary packet body, and size. It can also be used to transfer files under supported protocols.
- This tool is useful for testing firewall rules, spoofed port scanning, network performance, packet sizes, TOS (type of service), fragmentation, path MTU discovery, file transfer, traceroute with different protocols, remote OS fingerprinting, TCP/IP stack auditing, etc.
- As a very simple example, we can use hping2 to see if the host is running a web server as follows:

hping2 192.168.1.10 -S -p 80

- The above command will send a SYN packet to host 192.168.1.10 and display the response:



```
root@milliways:~ - Shell - Konsole <2>
Session Edit View Bookmarks Settings Help

[root@milliways root]# hping2 192.168.1.10 -S -p 80
HPING 192.168.1.10 (eth0 192.168.1.10): S set, 40 headers + 0 data bytes
len=44 ip=192.168.1.10 ttl=64 DF id=0 sport=80 flags=SA seq=0 win=32767 rtt=0.2 ms
len=44 ip=192.168.1.10 ttl=64 DF id=0 sport=80 flags=SA seq=1 win=32767 rtt=0.1 ms
len=44 ip=192.168.1.10 ttl=64 DF id=0 sport=80 flags=SA seq=2 win=32767 rtt=0.1 ms

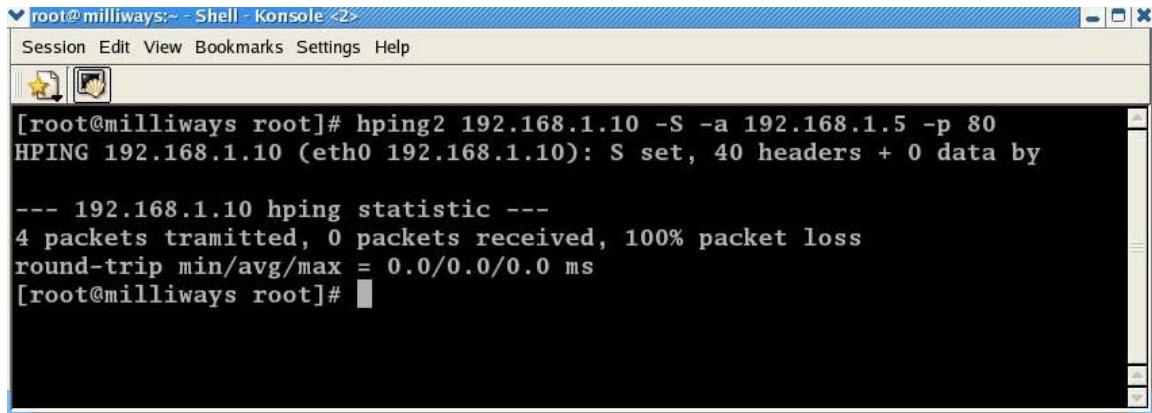
--- 192.168.1.10 hping statistic ---
3 packets tramitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.1/0.1/0.2 ms
[root@milliways root]#
```

- Notice that the return packets have the SYN/ACK flags set, which is the second stage of the TCP three-way handshake. This tells us that the service is running and responding to connection requests.

- We can also use hping2 to scan a remote system and spoof the source IP address. This is done by typing the following command:

*hping2 192.168.1.10 -S -a 192.168.1.5 -p 80*

- The above command will send SYN packets to 192.168.1.10 and give the originating IP as 192.168.1.5.

A screenshot of a terminal window titled "root@milliways:~ - Shell - Konsole <2>". The terminal shows the command `hping2 192.168.1.10 -S -a 192.168.1.5 -p 80` being executed. The output indicates that the command was successful, showing "HPING 192.168.1.10 (eth0 192.168.1.10): S set, 40 headers + 0 data by". Below this, a statistics summary is displayed: "4 packets transmitted, 0 packets received, 100% packet loss" and "round-trip min/avg/max = 0.0/0.0/0.0 ms". The prompt `[root@milliways root]#` is visible at the bottom of the terminal.

```
[root@milliways root]# hping2 192.168.1.10 -S -a 192.168.1.5 -p 80
HPING 192.168.1.10 (eth0 192.168.1.10): S set, 40 headers + 0 data by

--- 192.168.1.10 hping statistic ---
4 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
[root@milliways root]#
```

- Notice that there were no responding packets from the system. That is because the SYN/ACKs were sent to the spoofed IP address.

Example

- Let us assume that we want to design tests for some of the rules in a firewall ruleset.
- The first step would be to clearly outline the test for each rule, the tool used, and the expected results. The following is an abbreviated table summarizing the tests:

Rule #	Test Description	Tool Used	Expected Result	Pass/Fail
1	Drop all incoming packets that have both the SYN and FIN bits set together.	hping2	The firewall should drop the packet	Pass. Detailed results are attached.
2	Verify that SSH traffic is being mangled for minimize delay	Log on to the ssh server from and external ssh client	The <i>iptables</i> -L -v audit table should show that the traffic was mangled.	Pass. Detailed results are attached.

Test Case 1

- The following test was run to send packets with both the SYN and FIN bit set to the internal host:

```
# hping2 192.168.1.5 -s 1033 -p ++2048 -S -F -c 15
HPING 192.168.1.5 (eth0 192.168.1.5): SF set, 40 headers + 0 data bytes

--- 192.168.1.5 hping statistic ---
15 packets tramitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

- Before the test, the iptables log shows:

```
0 0 DROP tcp -- * * 0.0.0.0/0 0.0.0.0/0 tcp flags:0x3F/0x03
```

- Following the test, the iptables log shows:

```
15 600 DROP tcp -- * * 0.0.0.0/0 0.0.0.0/0 tcp flags:0x3F/0x03
```

- Notice that the packet count increased by 15, thus confirming that all 15 packets were successfully dropped by the firewall.

Test Case 2

- For this test an external client was used to log on to the server running the ssh service through the firewall.

- Before running test with an SSH client, iptables on the firewall reported:

```
0 0 TOS tcp -- * * 0.0.0.0/0 0.0.0.0/0 tcp spt:22 TOS set 0x10
0 0 TOS tcp -- * * 0.0.0.0/0 0.0.0.0/0 tcp dpt:22 TOS set 0x10
```

- After running the test, iptables reported:

```
20 3353 TOS tcp -- * * 0.0.0.0/0 0.0.0.0/0 tcp spt:22 TOS set 0x10
21 3036 TOS tcp -- * * 0.0.0.0/0 0.0.0.0/0 tcp dpt:22 TOS set 0x10
```

- The above confirms that iptables successfully mangled the SSH traffic.