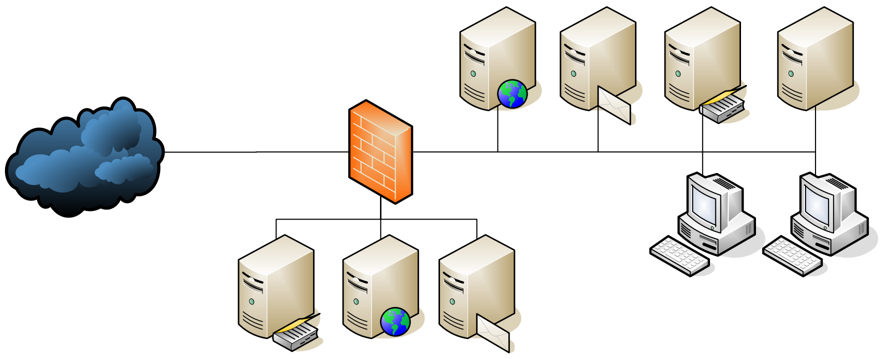
Firewall Configuration and Assessment

Goals of this lab:

- Get hands-on experience implementing a network security policy
- Get hands-on experience testing a firewall

Table of Contents

MAIN LAB



In this lab you will set up a Linux-based system as a firewall, according to a network security policy provided to you. The implementation will be done using a virtual Linux network designed to mimic a fairly typical corporate network scenario.

This document may appear quite long, but most of it is background and system information and tutorials. In addition to this document, you may find the APT lab from TDDI05 helpful: it covers how to install software in Debian/Gnu Linux, the operating system you will be using.

This lab assumes that you know how to use a Unix-like operating system from the command line. If you don't, you may find the lab quite difficult. It also assumes that you know the TCP/IP protocols well. Terms like "netmask", "port", and "prefix" should make sense to you. During these labs you *will* be expected to handle networking issues on your own. This lab does not provide step-by-step instructions; you are encouraged to come up with your own solutions.

Additional documentation

There is lots of information available about packet filtering in Linux. These are two recommended resources:

- The iptables manual page (access using `man iptables` on a Linux computer)

  Reference documentation for the iptables command; lists classifiers, targets, chains etc.

- The netfilter documentation page http://www.netfilter.org/documentation/index.html

  Lots of useful documentation. The packet filtering howto, iptables tutorial and iptables for newbies documents are probably the most relevant.

Part 1: User-mode Linux

Setting up the virtual machines

To set up the virtual machines for this lab, *run the following command.* (Where <*liu-id*> should be replaced with your actual LiU ID.)

```
tar -xzf /courses/TDDD17/lab/tddd17-fw.tar.gz  -C /home/<liu-id>/
```

This will copy the images for all UML hosts used in this lab into /home/<*liu-id*>/tddd17/fw.

Introduction to UML

This lab will be done using virtual machines: virtual computers that run as processes in the operating system. Virtual machines make it possible to run multiple operating system instances on the same physical hardware. The different instances are for all intents and purposes independent computers; they just happen to run on the same hardware.

Our virtual machines are implemented using user-mode Linux (UML), which is a port of Linux to the Linux system call interface (the hardware-dependent bits of Linux have been replaced by bits that make calls to the host operating system), and allows users to run any number of virtual systems ("UML instances", "UMLs", or "guests") under a normal Linux system (the "host") without the need for special privileges. The UML system also includes facilities for networking virtual machines.

From a user standpoint, a UML instance is just like a real machine. The work environment very much resembles a situation where a number of machines are connected to a console server and only accessible through a single text console or through the network.

Exercise 1: Using the start script

1.1.1 Run **`/courses/TDDD17/lab/start.sh --start all`** to start the UML instances. You will probably see a lot of messages in the terminal that say "`send_sock sending to fd 4 Resource temporarily unavailable`". These can be safely ignored. They do not impact the lab. Sometimes one or more UML instances get stuck during the boot phase. It could help to stop all instances (see below) and try again.

1.1.2 Try to log in to one of the UML instances. The root password for all virtual machines is *123*

1.1.3 Run **`/courses/TDDD17/lab/start.sh --exit`** to stop the UML instances.

1.1.4 Start the UML instances again.

Report: No report is required for this exercise.

Transferring files between the UML and the host

The easiest way to copy files from a UML instance to the host (and vice versa) is by mounting a directory on the host in the UML instance. The UML instances in this lab automatically mount the /home/<*liu-id*>/tddd17/fw/files/ directory as /host on the UML. Copying a file to this directory on the host workstation makes it available on the UML instance, and vice versa.

Exercise 2: Transferring files between the UML and the host

1.2.1 Start your UML instances again.

1.2.2 On the host, run `touch /home/<liu-id>/tddd17/fw/files/testfile`.

1.2.3 On any UML, do `ls /host`; the file testfile should be shown.

1.2.4 On any UML, do `cp /etc/passwd /host`; on the host, do `cat /home/<liu-id>/tddd17/fw/files/passwd` to display the contents of the password file you just copied.

Report: No report is required for this exercise.

Part 2: Linux networking basics

In this lab, the network is preconfigured. Nevertheless, it is probably useful to know some basic commands for manipulating the network configuration.

Interfaces

An interface is a logical device through which you can send network packets. Some interfaces correspond to physical devices (e.g. eth0 is usually an Ethernet card), while others may be logical devices (e.g. lo is a device that is used to communicate within the host, and sit0 is a device used to tunnel IPv6 inside of IPv4 datagrams). Each interface has a number of properties. Some, like the physical address of an interface, may be tied to the hardware, while others, such as the list of IPv4 addresses assigned to the interface, are not.

Interfaces are traditionally manipulated using the `ifconfig` command, but in modern Linux systems, the `ip` command is preferred. Both commands are located in the /sbin directory; if you are logged in as root, /sbin is in your PATH, so you can simply issue the commands as `ifconfig` or `ip`; if you are logged in as a normal user, you will have to specify the full path: `/sbin/ifconfig` and `/sbin/ip`, respectively. Since `ifconfig` is obsolete in Linux, we only cover `ip` here.

Basics of ip

The `ip` command, part of the iproute2 package, is the up-to-date method for manipulating the Linux network stack. Its syntax is quite different from most conventional Unix commands, although the style `ip` uses is becoming more and more common.

To list all link layer information (e.g. network interfaces and their physical attributes), use the command `ip link show`. On the same system as the previous command, this yields:

```
1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:04:61:5d:39:e8 brd ff:ff:ff:ff:ff:ff
```

Since the command is only showing link layer information, the IP addresses are not shown. To see IP addresses, use the command `ip addr show`. This yields:

```
1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:04:61:5d:39:e8 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.5/24 brd 10.0.0.255 scope global eth0
    inet 10.0.0.6/24 brd 10.0.0.255 scope global secondary eth0
       valid_lft forever preferred_lft forever
```

The general format of the `ip` command is `ip` *OBJECT COMMAND*, where OBJECT is the thing to manipulate: link (interfaces), addr (IP addresses), route (routes), rule (routing policy), and so forth; and COMMAND depends on the object. The `ip` command also contains (very) brief built-in help, accessed by specifying “help” as the object or command.

Some useful command lines include:

```
ip link set eth0 up
```

Brings up (enables) interface eth0.

```
ip link set eth0 down
```

Takes down (disables) interface eth0.

```
ip addr add 192.0.2.1/24 brd 192.0.2.255 dev eth0
```

Add the address of interface eth0 to 192.0.2.1, the netmask to 255.255.255.0 and the broadcast address to 192.0.2.255.

```
ip addr del 192.0.2.1/24 brd 192.0.2.255 dev eth0
```

Remove the address 192.0.2.1 from eth0 (netmask to 255.255.255.0, broadcast address 192.0.2.255.

```
ip route show
```

List the current routing table.

Other commands

There are a number of other important networking-related commands. These are not covered in detail here; consult the appropriate documentation instead.

Important note: There is a known problem with receiving ICMP reply packets over the virtual network used in the lab. Therefore, *you will not be able to ping external hosts on the internet from your UML instances.*

```
Ping
```

Sends ICMP echo packets to a particular host; used to test network reachability.

```
traceroute, tcptraceroute
```

Show the path packets take from the current host to a target. Regular traceroute uses UDP or ICMP, while tcptraceroute uses TCP.

```
netstat
```

Show information about the networking stack, such as active connections.

```
lsof
```

List open files. The -i option to lsof is useful to show which processes are using which network ports.

```
nmap
```

Network mapping tool, that can be used to scan a network for hosts, or hosts for open services. Includes some tricks to bypass firewalls.

```
tcpdump, wireshark, tshark
```

Listen to (possibly all) traffic on a network interface. The tcpdump command is non-interactive and mainly useful for recording network traffic. The wireshark command is graphical (in order to run it on the virtual machines you will have to connect to them using ssh) and can be used to capture and analyze traffic. The tshark command is a text-only variant of wireshark (similar to tcpdump, but with better facilities for analysis).

Exercise 1: Other commands

2.1.1 Start your UMLs and log in as root on your firewall (IP address 10.19.0.1).

2.1.2 On the firewall, use `tcpdump` to dump all network traffic on the external interface to a file. The following command will do that for you:

```
tcpdump -i eth0 -s 1500 -w /host/fw.pcap
```

The -i switch specifies which interface to capture on. The -s switch specifies how much of each packet to capture; here, we capture the entire packet (since the MTU of eth0 is 1500). The -w switch specifies a file to write the captured traffic to.

Leave `tcpdump` running through the next task.

2.1.3 Use `nmap` on the external UML host (ext) to scan the external interface of your firewall. The following command is suitable:

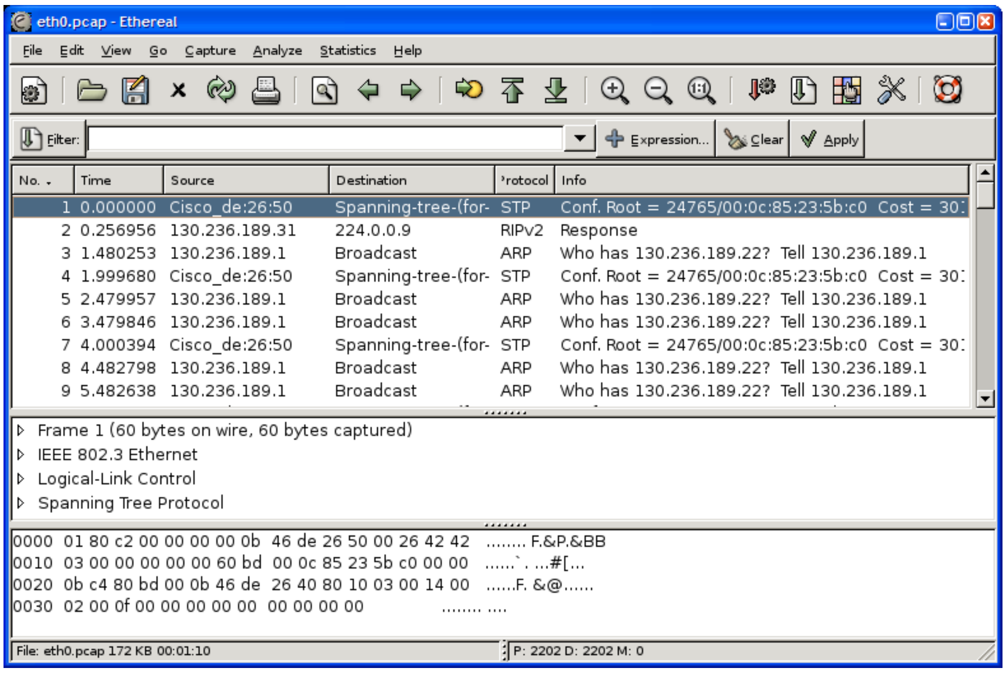
```
nmap -p 1-4096 -T Aggressive 10.19.0.1
```

The -p switch specifies which ports to scan. The -T switch specifies the timing of the scan; here we select an aggressive timing since we aren't concerned with minimizing bandwidth consumption or evading detection.

2.1.4 Stop `tcpdump` on the firewall, and load the captured traffic into `wireshark` on the host machine. Use the following command (replace *username* with your user name):

wireshark -r /home/<*liu-id*>/tddd17/fw/files/fw.pcap

The -r flag to wireshark tells it to load the specified file containing captured traffic. You should see a window that looks something like this:



```
No. . Time      Source           Destination          ›rotocol Info
    1 0.000000  Cisco_de:26:50   Spanning-tree-(for-  STP     Conf. Root = 24765/00:0c:85:23:5b:c0  Cost = 30:
    2 0.256956  130.236.189.31   224.0.0.9            RIPv2   Response
    3 1.480253  130.236.189.1    Broadcast            ARP     Who has 130.236.189.22?  Tell 130.236.189.1
    4 1.999680  Cisco_de:26:50   Spanning-tree-(for-  STP     Conf. Root = 24765/00:0c:85:23:5b:c0  Cost = 30:
    5 2.479957  130.236.189.1    Broadcast            ARP     Who has 130.236.189.22?  Tell 130.236.189.1
    6 3.479846  130.236.189.1    Broadcast            ARP     Who has 130.236.189.22?  Tell 130.236.189.1
    7 4.000394  Cisco_de:26:50   Spanning-tree-(for-  STP     Conf. Root = 24765/00:0c:85:23:5b:c0  Cost = 30:
    8 4.482798  130.236.189.1    Broadcast            ARP     Who has 130.236.189.22?  Tell 130.236.189.1
    9 5.482638  130.236.189.1    Broadcast            ARP     Who has 130.236.189.22?  Tell 130.236.189.1

▷ Frame 1 (60 bytes on wire, 60 bytes captured)
▷ IEEE 802.3 Ethernet
▷ Logical-Link Control
▷ Spanning Tree Protocol

0000  01 80 c2 00 00 00 00 0b  46 de 26 50 00 26 42 42   ........ F.&P.&BB
0010  03 00 00 00 00 00 60 bd  00 0c 85 23 5b c0 00 00   ......`. ...#[...
0020  0b c4 80 bd 00 0b 46 de  26 40 80 10 03 00 14 00   ......F. &@......
0030  02 00 0f 00 00 00 00 00  00 00 00 00              ........ ....

File: eth0.pcap 172 KB 00:01:10          P: 2202 D: 2202 M: 0
```

The bottom pane is a data dump of the selected packet; the middle pane is a dissection of the packet (click the arrows to show more detail), and the top pane is the list of captured packets.

2.1.5 You will see a lot of traffic not related to your scan. In particular, you can expect to see STP (Spanning Tree Protocol, ARP, and RIP traffic). Eliminate this from the display by entering the following string in the filter box above the captured traffic:

```
!stp && !arp && !rip
```

Hit enter to activate the filter. You should now see mostly TCP traffic, but there may be other kinds of traffic mixed in. You should now be able to determine what kind of packets `nmap` sends when scanning (when not run as root), and which order it scans ports in. If you change the filter to just `rip`, you can examine the RIP traffic on the network.

Report: No report is required.

Part 3: Linux packet filtering

In Linux, packet filtering is implemented through the netfilter framework, which defines a set of hooks in the kernel that user-defined programs can use to manipulate packets as they traverse the network stack.

The netfilter framework is used by the iptables software in Linux. iptables defines a set of tables that contain chains of rules that packets are matched against to determine how to process them. Users control iptables using the `iptables` command. The remainder of this section (and this lab) focuses on iptables.

Tables

When a packet is received, it is passed through a number of netfilter *tables*, which contain *chains*. Each chain is a list of rules that match properties of the network packet, and tell the kernel what to do with it. Each table is implemented by a kernel module (or may be statically linked in the kernel); each table then contains a set of chains, some predefined and some user-defined.

The standard tables are *nat*, *mangle*, and *filter*. There is also a *raw* table, which you can read more about elsewhere. The *nat* table is used for network address translation; the *mangle* table is used to modify packets (e.g. set TOS or DSCP bits), and the *filter* table is used for packet filtering. In this lab, you will mainly be using the *filter* table.

Chains and rules

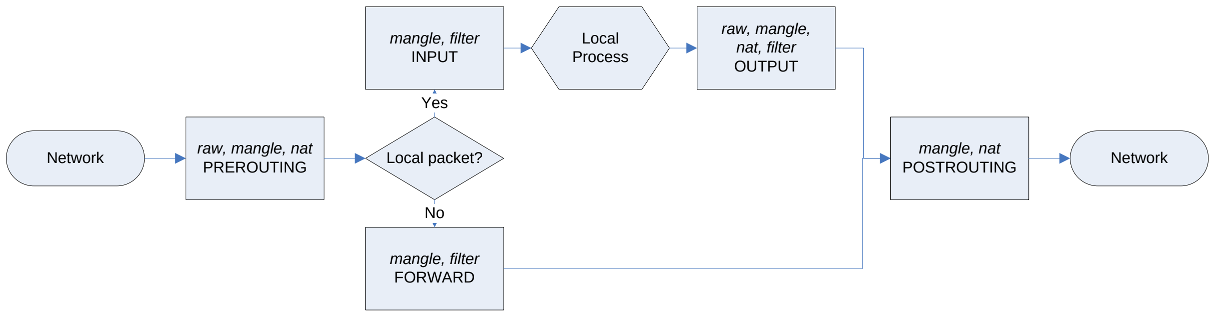
Chains are lists of rules that are applied in order to all packets that pass through the chain. Each rule consists of a set of classifiers that determines which packets the rule applies to, and a target, which determines what to do with the packet. Typical targets include DROP (the packet is thrown away), and ACCEPT (processing of the chain is aborted and processing of the packet continues in the kernel). There are a number of other targets, such as those implementing NAT, targets to modify packets, targets to process using a different chain, and so forth.

There are a set of standard chains, determined by the tables that have been loaded. Users can define additional chains, which behave much like functions in a programming language. User-defined chains are often used to encapsulate common behavior used in several standard chains.

The standard chains are PREROUTING (available in the *raw*, *mangle*, and *nat* tables), INPUT (*mangle* and *filter*), OUTPUT (*raw*, *mangle*, *nat*, and *filter*), FORWARD (*mangle* and *filter*), and POSTROUTING (*mangle* and *nat*).

PREROUTING is applied to all incoming packets before any routing decisions are taken. POSTROUTING is applied to all outgoing packets after all routing decisions have been taken. INPUT is applied to packets sent to the device (e.g. firewall) itself. OUTPUT is applied to packets generated by the device (e.g. the firewall) itself. FORWARD is applied to packets passing through the device (e.g. received on one interface and sent out on another).

The following flowchart shows the tables (italics) and chains (uppercase) that a packet flows through:



Built-in chains (PREROUTING, INPUT, OUTPUT, FORWARD, and POSTROUTING) also have a policy (user-defined chains do not), which is a target applied to any packet that reaches the end of the chain (because no classifier matched, or because no target aborted processing of the chain).

Classifiers

Classifiers are elements of the rules used to match network packets. For example, a classifier may match headers in the packet (e.g. IP, UDP, or TCP headers), information about the packet (e.g. which network interface it arrived on, or will be sent to), information computed from the packet (e.g. connection state), information added by the netfilter framework (e.g. packet or connection mark), and so on.

Some classifiers are standard, such as source and destination IP address, but most are implemented in match extension modules. Extension modules may be loaded automatically (e.g. when the IP protocol classifier is used) or manually using the -m switch to `iptables`.

Targets

The target of a rule specified what to do if all the classifiers in the rule match a packet. The standard targets are:

`ACCEPT`

The packet is let through. Processing of the current (stack of) chains is terminated, and the packet is passed on through the network stack.

`DROP`

The packet is dropped. Processing of the current (stack of) chains is terminated, and the packet is removed from memory.

`QUEUE`

If the kernel supports it, the packet is queued for processing in user space. This allows user-written programs to process packets as part of the netfilter framework.

`RETURN`

Processing of the current chain is terminated. If there is a calling chain, processing of that chain continues.

`Chain name`

The packet is passed to the named chain for further processing. If the packet is matched against a rule with a RETURN target in that chain (or fails to match any rule in a user-defined chain), processing of the current chain will continue.

Additional targets may be defined, depending on the configuration of the operating system kernel. Some examples of common targets (there are many more available) are:

`DNAT and SNAT`

Destination and network source address (and port) translation.

`LOG`

Log information about the packet to the system log.

`REJECT`

Reject a packet and notify the sender with an ICMP message. Also terminates packet processing and drops the packet.

Part 4: IP tables examples

All these examples use the filter table, which is used to filter packets (which is what a firewall is for, after all). Use of the nat table is covered below; use of the mangle and raw tables are outside the scope of this lab.

Adding, removing, and listing rules

Rules are added and removed from chains using the iptables command. The -A switch means append a rule to a chain, the -I switch means insert a rule (by default before the first rule), and the -D switch means delete a rule.

`iptables -A INPUT` *`rule`*

Append a rule to the INPUT chain in the filter table.

`iptables –D INPUT` *`number`*

Delete rule number *number* from the INPUT chain in the filter table. Rules are numbered from one and up.

`iptables –I INPUT` *`number rule`*

Insert a new rule as rule number *number* in the INPUT chain of the filter table. Rules are numbered from one and up. If no number is given, insert as the first rule in the chain.

`iptables -F INPUT`

Delete all rules from the INPUT chain in the filter table (F stands for "flush").

`iptables -n -L INPUT`

Lists all rules in the INPUT chain, displaying addresses and ports numerically.

`iptables -n -v -L INPUT`

Verbose rules listing, including how many packets (and bytes) have matched each rule.

`iptables -N` *`chain`*

Create a user-defined chain named *chain*, that can be called from other chains using the chain name as the target.

To manipulate chains in other tables than the filter table, use the -t *table* option to `iptables` before any -A, -D, or other switch.

Writing your rules in a file

It is very convenient to write your rules in a file and then simply execute the file, rather than entering the rules one-by-one at the command line. A good approach is to start the file with commands that clear all current rules (effectively resetting iptables). On the host machine, create a file called `exercises.sh` (using any editor such as emacs, vi, nano, etc.) and place it in `/home/<liu-id>/tddd17/fw/files/`. Now enter the following into this file:

```
#!/bin/sh
echo "Flushing iptables rules..."
sleep 1
iptables -F
iptables -X
iptables -t nat -F
iptables -t nat -X
iptables -t mangle -F
iptables -t mangle -X
iptables -P INPUT ACCEPT
iptables -P FORWARD ACCEPT
iptables -P OUTPUT ACCEPT

echo "Setting iptables rules..."

# Enter your rules here

echo "Setting iptables rules... done"
```

This file is now available on your firewall. To execute the file (and thereby loading all the rules into iptables) run the following command on your firewall:

```
sh /host/exercise.sh
```

Exercise 1: Adding, removing, and listing rules

4.1.1 Start your UMLs and log in as root on the firewall.

4.1.2 Insert a rule that blocks ssh access to the firewall by using the following command:

```
iptables –A INPUT –p tcp --dport 22 –j DROP
```

4.1.3 Verify that you cannot connect to the firewall using ssh from the external host by using the following command:

```
ssh –v –l root 10.19.0.1
```

The -v switch causes `ssh` to show debugging information. If your firewall rule is successful, it should never get further than “Connecting to…”. You can terminate the command by pressing Ctrl+C.

4.1.4 Run the same `nmap` command as before, what has changed in the results?

4.1.5 Now try and use `REJECT` instead of `DROP` in your firewall rule, run `nmap` again, what has changed this time?

4.1.6 List the INPUT chain with the -v switch, and verify that the packet counters seem reasonable, considering the connection attempts you have made.

Report: No report required.

Writing rules

Writing good rules is at the core of firewall configuration. An iptables rule matches a packet if all the classifiers in the rule match the packet. There is no provision for combining classifiers with arbitrary logic. There is a small set of built-in classifiers, and a large number of classifiers implemented through match extensions. The most common built-in classifiers are:

`--protocol` *`proto`*

Matches if the IP protocol of the packet is *proto*. The protocol can be specified numerically, or with one of the strings “tcp”, “udp”, or “icmp”. Specifying a protocol automatically loads the match extension for that protocol. Shorthand: `-p`.

`--source` *`CIDR`*

Matches if the source IP address of the packet has prefix *CIDR*. For example, `--source 192.0.2.0/24` will match 192.0.2.1 (and 255 other addresses). Shorthand `-s`.

`--destination` *`CIDR`*

Matches if the destination IP address of the packet has prefix *CIDR*. For example, `--destination 192.0.2.0/24` will match 192.0.2.1 (and 255 other addresses). Shorthand: `-d`.

`-in-interface` *`if`*

Matches if the network interface the packet arrived on is *if*. Shorthand: `-i`.

`--out-interface` *`if`*

Matches if the network interface the packet is to be sent to is *if*. Shorthand: `-o`.

All match extensions are documented in the manual page for the `iptables` command. Some need to be loaded manually using the -m switch to `iptables`; others are loaded automatically, such as the match extensions for various IP protocols. For the tcp and udp protocols, these are some of the most common match extensions:

`--source-port` *`port`*

Matches if the source port of the packet is port. Shorthand: `--sport`.

`--destination-port` *`port`*

Matches if the destination port of the packet is port. Shorthand: `--dport`.

See the `iptables` documentation for additional possibilities.

Exercise 2: Writing rules

4.2.1 Write a rule that accepts ssh traffic (TCP, port 22) arriving through interface eth0.

4.2.2 Write a rule that accepts traffic to a DNS server (UDP, port 53) from any address in the range 10.0.0.0-10.0.0.255. (Hint: that's an entire network.)

4.2.3 Write a rule that accepts traffic from any address in the range 10.0.0.1 to 10.0.0.6, inclusive. (Hint: that's not an entire network; but there is a suitable extension match that you can read about in the man page for `iptables or  search for` "*`iptables ip range`*").

Report: No report required.

Setting chain policy

All built-in chains have a policy (user-defined chains don't), which is the target used when no rule in the chain matches a packet. The policy is set using the -P command-line option to `iptables`.

```
iptables -P INPUT DROP
```

Sets the policy of the INPUT chain in the filter table to DROP.

```
iptables -P OUTPUT ACCEPT
```

Sets the policy of the OUTPUT chain in the filter table to ACCEPT.

The policy of a built-in chain implements the default policy of the firewall. For example, if the network policy states that traffic not explicitly permitted shall be dropped, then it is appropriate to use DROP as the policy for the corresponding chain.

An alternative to setting the policy of a chain is to terminate the chain with one or more rules that matches all packets. For example:

```
iptables -A FORWARD -j REJECT --reject-with icmp-net-unreachable
```

This will cause all packets to be rejected with an ICMP network unreachable message. One possible reason for using a rule rather than using the chain policy is to implement network policy that states what to do when e.g. the rule set fails to load. For example, network policy may state that traffic is to be disallowed by default, but if the rule set fails to load, all traffic should be allowed.

Report: No report required.

Matching new connections (optional, only read if you are interested)

It is often necessary for a firewall rule to match packets that establish new TCP connections, distinguishing them from packets that belong to already established connections. The most common case is probably to let packets belonging to existing connections pass through the firewall. There are two ways to distinguish packets belonging to new and to established connections in Linux: looking at the TCP header flags, or using the connection tracking feature of iptables.

Using TCP flags

Connection attempts can be recognized by examining the SYN flag of a TCP packet. If the SYN flag is not set, then the packet cannot possibly establish a new connection, and must belong to an existing connection. If the SYN flag is set and ACK or RST are also set, then the packet is a response to a SYN packet, and can also be considered part of an established connection.

Linux iptables has a --syn switch designed for this task; there is also a general-purpose --tcp-flags switch. The following two commands are equivalent:

```
iptables -A FORWARD -p tcp --tcp-flags SYN,RST,ACK SYN -j DROP
iptables -A FORWARD -p tcp --syn -j DROP
```

A more complete example is shown below, in which new connections to ssh and HTTP are accepted, but all other new connections are dropped. All packets that do not represent new connections are then accepted:

```
iptables -A INPUT -p tcp --syn --dport 22 -j ACCEPT
iptables -A INPUT -p tcp --syn --dport 80 -j ACCEPT
iptables -A INPUT -p tcp --syn -j DROP
iptables -A INPUT -p tcp -j ACCEPT
```

Using connection tracking

Another option is to use the connection-tracking feature of Linux. Connection tracking maintains state information about each connection to or through the system. By matching connection state rather than just flags, it is possible to distinguish packets belonging to new and to establish connections. The state-matching module of iptables is the simplest way to accomplish this. It accesses the connection tracking state for the connection a packet belongs to; the state is NEW for new connections (i.e. connection attempts), ESTABLISHED for established connections; INVALID (if the packet does not conform to expected behavior of the connection); or RELATED (if the packet does not belong to, but is somehow related to an established connection). Typically, the following rule is used to accept packets belonging to established connections:

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

New connections are not accepted by this rule (and need to be dropped either using another rule, or by the chain policy). For example, the following example shows how to use connection tracking to accept ssh and HTTP connections only (similar to the example above):

```
iptables -A INPUT -p tcp -m state --state NEW --dport 22 -j ACCEPT
iptables -A INPUT -p tcp -m state --state NEW --dport 80 -j ACCEPT
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -j DROP
```

Part 5: Network Address Translation

Usually when a packet traverses an IP network, it is left mostly alone. The routers on the path of the packet are only concerned with sending the packet on to the next hop along the path. The only part of the packet they *should* be altering is the TTL field (and the checksum) in the IP header.

When a router rewrites the source or destination address of a packet, it is performing network address translation (NAT). It may also alter the source or destination port in TCP and UDP packets, in which case it is doing network address and port translation (sometimes known as NAPT). Typically it will remember what translations it has performed, and when reply packets are sent, it will reverse the translation.

Why and how

NAT is most commonly used to connect systems using RFC1912 (private, unrouteable) address space to the Internet. Private addresses cannot be used on the Internet, so a router that knows about the private address space must substitute some other, public, address for the source address in the packet.

In the simplest case a unique public address is mapped to each (presumably private) source address. This is called 1:1 NAT, and requires nothing more of the router than to rewrite the source address of outgoing packets and the destination address of incoming packets.

More commonly NAT is used to map a set of (presumably private) source addresses to a single public address. The challenge here is how to figure out which reply packet goes to which private address. Essentially, the router must ensure that reply packets (which will be addressed to the router) will contain enough information to uniquely identify their real destination. Leaving non-TCP/UDP packets aside for the moment, this is accomplished by examining the source and destination addresses *and* ports of reply packets. When a packet is sent out (from a private address) through a NAT router, it records the real source address and port, and associates that with the rewritten source address and port, and destination address and port. When a reply packet is processed, the router matches the packet's destination address and port against the recorded source address and port, and the packet's source address and port against the recorded destination address and port. This uniquely identifies the real source address and port, and that information is placed in the packet, which can then be sent to its true destination.

Since all packets in a connection must have the same source and destination addresses and ports, the process of recording the real source address and port, and mapping it to a rewritten source address and port, can only be done for the first packet in a connection; subsequent packets belonging to the same connection must use the same mapping. For stateful protocols like TCP, this can be done reliably, since the router can use the same mechanisms the source and destination use to keep track of the connection's state. For connectionless protocols, such as UDP, routers must find a way to create a "virtual" connection, by examining e.g. timing information or application data.

NAT with iptables

NAT is supported by iptables through the nat table, and the SNAT, DNAT, and MASQUERADE targets. SNAT (Source NAT) implements translating the source address of outgoing packets (and the appropriate transformation of reply traffic), DNAT (Destination NAT) implements translating the destination address of incoming traffic (and the appropriate transformation of reply traffic). MASQUERADE is similar to SNAT, but is used in situations where the target address of the translation is unknown (e.g. when it is set using DHCP).

For example, the following command causes all traffic coming in on interface eth0 from the network 10.0.0.0/8 to be translated.

```
iptables -t nat -A POSTROUTING -i eth0 -s 10.0.0.0/8 –j SNAT --to-
source 192.0.2.1
```

This command could be used on the same NAT router to forward TCP connection attempts to port 80 on 192.0.2.1 to port 8080 on the host 10.0.0.1:

```
iptables -t nat -A PREROUTING –p tcp -d 192.0.2.1 --dport 80 -j DNAT -
-to-destination 10.0.0.1:8080
```

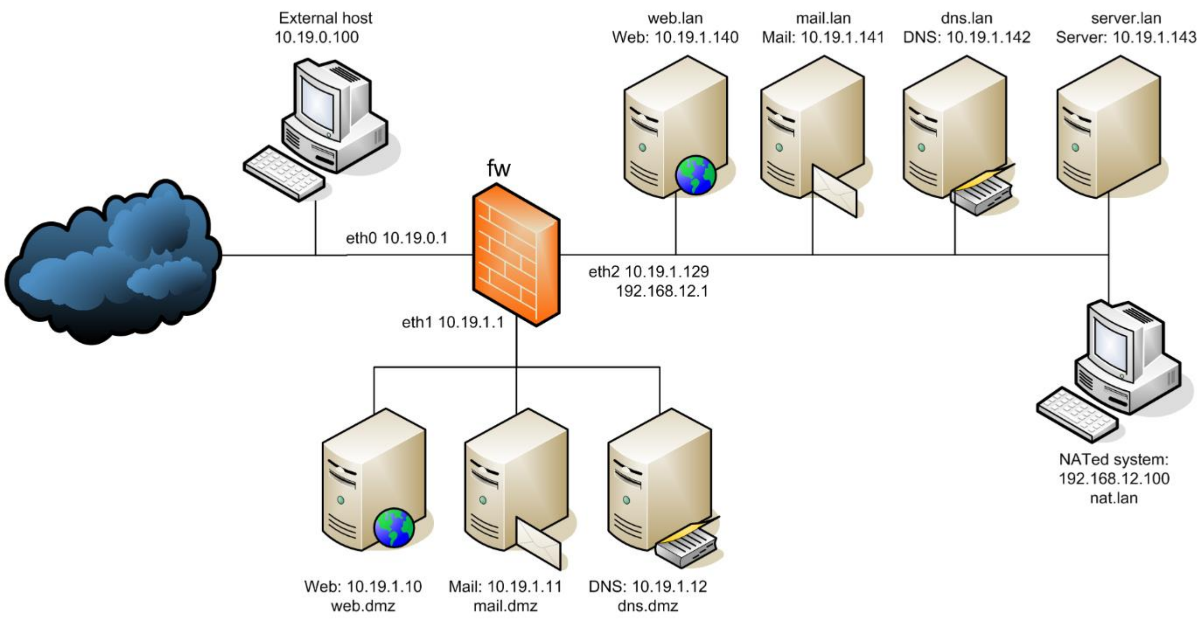
The following commands set up 1:1 NAT for a single address. Unfortunately, because of they way NAT is implemented by iptables, the router will still maintain state data for all connections, which consumes memory and time:

```
iptables -t nat -A POSTROUTING -s 10.0.0.2 -j SNAT --to 192.0.2.2
iptables -t nat -A PREROUTING -d 192.0.2.2 -j DNAT --to 10.0.0.2
```

Stateless 1:1 NAT is currently not supported in Linux (it used to be, but didn't work properly). Then again, if that is what you think you need, there are other solutions than NAT.

Part 6: System description

The firewall you will be configuring is fairly typical. It consists of a single firewall with three ports: one for the Internet; one for a “DMZ”, which houses services accessible from the Internet; and one for an office LAN, which houses internal services, workstations and such systems.



Each system has a separate UML instance with a single IP address. The appropriate services are configured (to some extent) on each of your servers. The web server is dhttpd, a small and very simple server; the mail server is Postfix; and the DNS server is BIND 9. All hosts use the internal DNS server to look up names.

The DNS servers are configured with zones. Using `dig` or `host`, you can look up names like ida.liu.se, if you send the query directly to the name server.

Host	Interface	Address	Purpose
fw	eth0	10.19.0.1	External (Internet) interface
	eth1	10.19.1.1	DMZ interface
	eth2	10.19.1.129	Internal (LAN) interface
web.dmz	eth0	10.19.1.10	Web server (dhttpd)
mail.dmz	eth0	10.19.1.11	SMTP server (Postfix)
dns.dmz	eth0	10.19.1.12	DNS server (BIND 9)
web.lan	eth0	10.19.1.140	Web server (dhttpd)
mail.lan	eth0	10.19.1.141	SMTP server (Postfix)
dns.lan	eth0	10.19.1.142	DNS server (BIND 9)
server.lan	eth0	10.19.1.143	Workstation/server (no services listening)
nat.lan	eth0	192.168.12.100	Workstation
ext	eth0	10.19.0.100	Test system on the outside of the network

Tools for testing the web server

If you want to access the web server from one of your virtual machines then the command-line tools are `curl` and `wget`. For example:

```
wget http://10.19.1.140/
```

If you need more control, for example if you want to control the source address or source port, `socat` (see below) is the best choice (you'll have to type in an HTTP request manually).

If you want to be able to download a file from your web server, create the file in /var/www on the web server.

Tools for testing the name server

The two tools most commonly used when working the DNS are `host` and `dig`. Both are pure DNS tools – they will not query any other information source – and both allow you to manipulate DNS queries at a fairly low level. The `host` utility is convenient to run simple queries, while `dig` is appropriate when more information and more control is required.

The simplest query possible with `host` is to request the address corresponding to a particular name. It is possible to direct queries to a particular name server.

`host` *`name nameserver`*

Query *nameserver* for the address corresponding to *name*. Whenever possible, use an IP address, not a name, for *nameserver*. If the *nameserver* parameter is omitted, query the default name server, as specified in /etc/resolv.conf

With additional command line options, `host` can be used to extract other information from the domain name system. The full details are in the man page for `host`. These are some of the more common and useful options:

`host –a` *`name`*

Query for any resource record associated with *name*.

`host -l` *`zone`*

List the entire contents of *zone*. This option attempts to initiate a zone transfer, which may not be permitted by the queried name server.

`host –u` *`query`*

Use TCP for the query and reply.

`host –d` *`query`*, host `–dd` *`query`*

Display additional debugging information, such as the exact DNS queries sent.

Exercise 1: Using host

6.1.1 Use `host` to query your external name server using TCP using the following command:

```
host -T web.dmz.tddd17.ida.liu.se 10.19.1.12
```

Repeat with your internal server.

6.1.2 Use `host` to query your external name server using UDP using the following command:

```
host web.dmz.tddd17.ida.liu.se 10.19.1.12.
Repeat with your internal name server.
```

Report: No report is required.

Testing the mail servers

If you are trying to test if somebody from host A can send mail to the mail server on host B, a simple way to do so is by using netcat. The following command will open a connection to the SMTP port on *hostname*, allowing manual interaction with a mail server running at that host:

nc *hostname* 25

A more advanced alternative is socat, as explained below. (Note that, if you try to listen on port 25 using `socat` on host B, it is likely that the mail server is already using this port, and your `socat` command will fail. To stop the mail server from running, execute the following command:

systemctl stop postfix)

Using socat for generic network testing

For generic network testing, both sending and receiving, one of the best choices is to use `socat`. This is a program that connects data sources with destinations in a very flexible way. You can easily use `socat` both as a network server and as a client.

The manual page for `socat` can give you all the nasty details, but here are some quick recipes (there are more near the end of the manual page):

```
socat tcp4-listen:8080,reuseaddr,fork stdio:
```

Listen for TCP (over IPv4) connections on port 8080, and output anything from that connection to the terminal. Change tcp4 to udp4 to receive UDP packets instead.

```
socat open:data.txt tcp4:192.0.2.1:80,bind=192.0.2.10:5533
```

Send the contents of the file data.txt to TCP port 80 on 192.0.2.1 (change tcp4 to udp4 in order to send using UDP). The (entirely optional) bind parameter says to connect from source address 192.0.2.1, port 5533 (the source address must be configured on some network interface).

Many people use the netcat (`nc`) command for things like this, but `socat` is *far* more flexible.

The test system

The test system, ext, is included in the lab setup to give you a platform from which to run tests from outside the network.

Part 7: Firewall implementation

Exercise 1: Configure the firewall

7.1.1 For each requirement below, specify it in terms of the system and network (i.e. address instead of host, interface or address range instead of DMZ, Internet or LAN, and so forth).

7.1.2 Write a firewall configuration for your firewall (i.e. you are not to configure firewalls on the individual hosts) that meets all the requirements above.

Report: Your firewall configuration (all the rules, in the correct order, in a file as previously described), and documentation mapping each requirement to the corresponding (set of) firewall rule(s). We will run the file on our system without modification, so make sure it is syntactically correct before handing it in.

Firewall requirements

General policy

1. Hosts on the Internet MUST NOT initiate connections to hosts on the LAN or DMZ, other than explicitly provided by this policy.
2. Hosts on the DMZ MUST NOT initiate connections to hosts on the LAN or the Internet, other than explicitly provided by this policy.
3. Hosts on the LAN MAY initiate connections to the Internet and DMZ, other than explicitly prohibited by this policy.

DNS

4. Hosts on the Internet MUST be able to send DNS queries to the external DNS server.
5. Hosts on the DMZ MUST be able to send DNS queries to the internal DNS server.
6. The external DNS server MUST be able to respond to DNS queries.
7. The external DNS server MUST be able to send DNS queries to hosts on the Internet.

Mail

8. Hosts on the Internet MUST be able to connect to SMTP on the external mail server.
9. The external mail server MUST be able to connect to SMTP on the internal mail server.
10. The internal mail server MUST be able to connect to SMTP on mail servers on the Internet.
11. Hosts on the LAN other than the internal mail server MUST NOT be able to connect to SMTP on hosts on the Internet or the DMZ.

Web

12. Hosts on the Internet MUST be able to connect using HTTP to the external web server.

Firewall

13. The firewall MUST be able to communicate with itself.
14. The firewall MUST accept RIP traffic from the Internet.
15. The firewall MUST accept ssh connections from the LAN.
16. The firewall MUST NOT accept any other traffic from the LAN, DMZ, or Internet.

Other

17. The firewall must implement source NAT for LAN hosts in 192.168.12.0/24.
18. IPSec connections MUST be permitted from the Internet to the LAN.
19. Inappropriate ICMP types MUST NOT be permitted from the Internet to the LAN.

Notes on ICMP

Some firewall recommendations say to block all ICMP. That's a really, really bad idea. In order to figure out what ICMP types to block, read about all the types, and determine which ones are necessary to ensure that your network functions properly. Note that the types of ICMP messages appropriate for outgoing traffic may be different than the ones appropriate for incoming traffic.

Part 8: Assessing the firewall (optional)

This section is optional. You do not need to do it at all, unless you really want to.

Implementing a firewall rule set is only part of building a good firewall. You will perform two types of assessment of the firewall you've built: the first is a non-adversarial test, designed to ensure that the firewall meets all requirements; the second is an offensive test, attempting to bypass the firewall in some way.

Exercise 1: Assessing your firewall

8.1.1 Test your firewall to ensure that it meets all stated requirements. Your test must include a scan of which hosts and ports are reachable on the DMZ and LAN from the Internet, on the LAN from the DMZ, and on the DMZ from the LAN. Additional tests are also needed.

Report: The exact commands used to test the firewall, your interpretation of the results of those commands, and a protocol showing how the tests demonstrate that the requirements have been met.