TÉCNICO LISBOA

Computer Networks — 2021/22		Assignment:	Lab 5
Configuring a Firewall & NAT with IPTables		Issued:	2021-12-13
Firewall / NAT		Submission Due:	2021-12-16
Authors:	João Romeiras Amado Prof. Luis D. Pedrosa	Comments Due:	2021-12-19
		Version:	1.0

Submission note. You will need to later submit all the answers to this lab's questions in Moodle. You should create a PDF with all the answers, and submit it into Moodle as an attachment to the submission form.

1 Goals

- Explore different firewall rule configurations using IPTables.
- Configure network address translation (NAT) on a given topology.
- Learn how to implement port forwarding.

2 Exercises

2.1 IPTables configuration and firewall

IPTables is a Linux utility program that allows a system administrator to define and configure rules for processing network packets in the Linux kernel, to handle traffic that departs from, arrives at, or is forwarded through the system. IPTables rules form policy chains that provide a sequence of rules that are checked in order. Each rule indicates a match pattern to check whether it should apply to a given packet. When a packet matches, you can also specify an action to apply including, e.g. dropping or modifying the packet.

In this lab you will explore how to use IPTables to implement firewall policies that define which traffic should be allowed and which should be blocked. We'll also explore Network Address Translation (NAT), a technique that modifies packets to allow multiple computers to share a single public IP address or to direct different TCP/UDP ports to specific servers.

IPTables allows users to manipulate *tables*. Each table contains several *chains* – sequences of rules that are executed in order for each packet. You will be manipulating these tables and chains by adding new rules to implement some common policies.

Though you can also create user-defined chains, in this lab we'll be focussing on the built-in ones. The `filter` table is used to specify policies that allow or drop packets. This table has three important built-in chains: `INPUT` to process packets meant for local delivery, `OUTPUT` to process outgoing packets that were generated locally, and `FORWARD` to process packets that pass through the local system.

As a network packet is processed, the kernel traverses the relevant chain and checks each of the rules in turn until one matches, at which point, the corresponding action is employed. If no rules apply, the chain's *default policy* is followed to process the packet.

Hint. Explore the IPTables command syntax using `man iptables`.

Computer Networks — 2021/22		Assignment:	Lab 5
Configuring a Firewall & NAT with IPTables		Issued:	2021-12-13
Firewall / NAT		Submission Due:	2021-12-16
Authors:	João Romeiras Amado Prof. Luis D. Pedrosa	Comments Due:	2021-12-19
		Version:	1.0

Consider the following example, that drops all incoming TCP packets:

```
iptables -I INPUT -p tcp -j DROP
```

The `-I` flag *inserts* a rule at the beginning of a particular chain (in this case, the INPUT chain). An alternative would be the `-A` flag, which *appends* a rule to the end of a given chain. We are omiting the `-t` option, thus manipulating the default table `filter`. Later we will use `-t nat` to edit the *nat* table. The `-p` flag defines the packet *protocol*. If this option is not specified, the value `all`, which corresponds to all protocols, is assumed by default.

The `-j` flag specifies the action to *jump* to should the rule criteria match. In this example, any incoming TCP packets should be *dropped*. Common actions are `ACCEPT` or `DROP`, which stop packet processing and proceed with *allowing* or *blocking* the packet respectively.

Though the full list of filtering options is described in the manual, in this lab we'll be frequently using the following:

`-s` Specifies the *source* IP address.
`-d` Specifies the *destination* IP address.
`--sport` Specifies the TCP/UDP *source port*.
`--dport` Specifies the TCP/UDP *destination port*.
`-i` Specifies the *incoming* network interface.
`-o` Specifies the *outgoing* network interface.

In addition to inserting or appending rules, as above, you can also *list* a chain's current rules:

```
:~$ iptables [-t <table>] -L [chain]
```

Or delete them all (*flush*):

```
:~$ iptables [-t <table>] -F [chain]
```

> **Hint.** During this laboratory, you will find it convenient to periodically flush chains, to reset things between exercises.

Firewalls can either be configured on network devices that filter traffic between connected networks, or on individual machines to filter traffic that they send and receive themselves.

Let's get started:

1. Execute the command `git pull origin master` in `/home/rc/lab-files`.
2. In the CORE emulator, activate the following topology: `iptables-1.imn`.

Our aim is to configure IPTables rules to enact policies at the various nodes. Let's start by protecting `n3`.

Q1. Write an IPTables rule to block all incoming traffic for `n3`.

Test the rule by pinging `n3` from another node.

Having ping is useful for debugging networks. Let's fix this by allowing incoming pings (recall from previous labs in wireshark: ping uses the ICMP protocol).

Q2. With the previous IPTables rule still active, write an IPTables rule to only allow incoming pings for `n3`.

Try the ping command again and confirm that it works once more.

Now let's prepare `n3` to run some servers. We'll allow the following:

- Incoming traffic on TCP port 5000, from all IPs.
- Port $NUM_LETTERS, only from connections coming from host `n7`.

Where $NUM_LETTERS is equal to the number of letters in your first + last name, multiplied by 100 (E.g., João Amado: 9*100=900).

Q3. Write the IPTables rules that correspond to this firewall configuration.

> **Tip.** iperf can be used to quickly run a server and check if it receives traffic. Read the manual to learn how to configure the protocol and port it uses.

`nmap` is a powerful command-line tool that can be used to scan IP addresses and ports. When scanning a host, `nmap` tries to start a connection on each port to see if it responds. A port can thus be *OPEN* if traffic is allowed and a server is running on that port, *CLOSED* if the firewall allows traffic, but no server is running on the port, or *FILTERED* if traffic is blocked altogether.

> **Warning.** Port scanning is often one of the first steps an attacker takes to find vulnerabilities on a host or in a network. Many networks thus have intrusion detection systems that flag port scans and may take action. As such, don't scan hosts without permission. Avoid performing a lot of scans from the University network, as our network admins may cut you off the network and send you a nastygram. Feel free to do port scans in CORE, as those never leave your machine.

By default, `nmap` checks a variety of commonly used ports: `nmap <host>`. You can, however, list specific ports to scan. For example, `nmap 1.2.3.4 -p 1000,1001` would check the status of ports 1000 and 1001 for address 1.2.3.4. For more details regarding nmap usage, you can use `man nmap`.

Q4. Scan the server using `nmap` to confirm that the intended configuration is active. Take a screenshot showing the `nmap` output.

2.2 NAT

NAT (network address translation) is a mechanism that modifies packet headers to either change where it goes (DNAT – destination NAT) or to mask where it came from (SNAT – source NAT). The most common form of NAT is used to change the source IP address so

a household can have multiple devices that share a single public IP address. You likely have one of these at home in your WiFi router.

NAT is typically done statefully, in that the translation is processed for the first packet in a connection and stored in a *connection table* to remember how to translate subsequent packets for the same connection. The IPTables `nat` table is used to process this first packet per connection, but not the rest. The `filter` table sees all packets and can even access the connection state when deciding to ACCEPT or DROP packets.

Consider a typical Internet connection sharing NAT for a household. Connections coming from inside the house reach the NAT with an internal IP address as its origin. Other hosts across the Internet cannot reach private addresses, so the NAT must change the source IP address of traffic leaving the house to the *NAT's public IP address*. The NAT must also store in its connection table where this connection came from (based on the TCP/UDP ports). This way, when traffic returns from the Internet back to the NAT, it knows how to change the destination IP address so the packet reaches the original internal host. To enact such a policy, we use the `SNAT` target, as follows: `-j SNAT --to-source $ADDRESS`. Analogously, to change the destination IP address, you use the `DNAT` target: `-j SNAT --to $ADDRESS:$PORT`.

As we are changing packets in flight, this can affect the route they end up taking, particularly if we change the destination IP address. As such, it's important for the kernel to know when it should make the decision of where to send the packet (based on the destination address). With this in mind, the `nat` table has two built-in chains: `PREROUTING` and `POSTROUTING`. Packets from new connections are first processed in the `PREROUTING` chain. At this stage, the kernel has not yet decided where the packet may go, so it doesn't make sense to filter on the outgoing interface (`-o`). That said, you can use DNAT to modify the destination address safely.

After processing the packet in the `PREROUTING` chain, the kernel performs the *routing decision* and determines which network interface it will send the packet out on. Now the packet is processed in the `POSTROUTING` chain. You should no longer change the destination IP address, as that would invalidate the routing decision, but you can now filter based on the outgoing interface (`-o`) and perform SNAT.

Consider that we wish to allow `n1`, `n2`, and `n3` to all share a common Internet connection via `n4`. In other words, `n4` should perform SNAT for all traffic heading towards `n5`.

Q5. Write the IPTables rules that correspond to this SNAT configuration.

Q6. Perform a ping from `n2` to `n7`. Take two screenshots of wireshark showing the ping requests and responses at `n2` and `n7`. Which node does `n7` look like its talking to? And `n2`?

2.3 Port Forwarding

Port forwarding is the process of forwarding traffic requests for a specific port to a different host.

Consider now that we want to configure port forwarding from `n4` (IP 10.0.3.1, port 80) to `n3` (port 80), specifically. When active, any connection made to `n4`'s IP address on this port should be seamlessly forwarded to `n3`.

Q7. Which IPTables chains must be configured to implement port-forwarding? Explain.

Q8. Write the IPTables rules to configure this port forwarding from `n4` to `n3`.

Q9. Use iperf to run a server on `n3` and test your configuration with a client on `n7`. Take two screenshots of wireshark at `n3` and `n7`, showing the port forwarding in action. Does `n3` realize that port-forwarding is happening?