Firewalls

Network Infrastructures Lab Sessions

SAPIENZA
UNIVERSITÀ DI ROMA

Instructors:
- Pietro Spadaccino

Today

- Firewalls with Iptables

Further readings:

- Linux iptables Pocket Reference, Gregor N. Purdy

Firewalls

- A *firewall* is a network security system that **monitors, controls, filters** and **modifies** incoming and outgoing network traffic based on predetermined security rules
    - For example, firewalls could allow or block traffic on some specific TCP/UDP ports or from specific IP addresses
- Advanced firewall rules may include constraints on traffic flow rate, etc.
- Usually firewalls have facilities implemented at the kernel level, for performance reasons

Netfilter and iptables

- **NETFILTER** is a framework that provides hook handling within the Linux kernel for intercepting and manipulating network packets
- **iptables** is the userspace frontend of NETFILTER, i.e. is the userspace application used to configure NETFILTER

We are going to use **iptables.**

From now on, when we will talk about iptables, in in reality the backend facilities are offered by NETFILTER, just keep that in mind

iptables

- iptables enables the user to manage NETFILTER "hooks": a hook is an entry point within the linux kernel IP networking subsystem that allows packet mangling operations: **intercepted** by a **matching rule** and **processed** by a **configured action**:
    - Common matches: protocol, source/destination address or interface, source/dest port…
    - Common targets: ACCEPT, DROP, REJECT, MASQUERADE, …

Examples:

- **Accept** packets of **TCP port 22 coming from interface eth1**
    - `iptables -A INPUT -p tcp --dport 22 -i eth1 -j ACCEPT`
- **Drop** packets **routed to subnet 192.168.1.0/24**
    - `iptables -A FORWARD -d 192.168.1.0/24 -j DROP`
- **Redirect** packets of **TCP directed to local port 80 to port 8080**
    - `iptables -t nat -A PREROUTING -p tcp --dport 80 -j DNAT --to-destination 127.0.0.1:8080`

Tables and Chains

We can write **rules** in iptables and put them inside **chains**. Chains are contained into **tables**.

When a packet goes through the firewall, it assumes several “**states**” inside the networking stack, e.g. packet reception, before routing takes place, packet directed to localhost, packet directed elsewhere, after routing takes place, etc…

In each one of these states, iptables **calls** different tables and chains: it tries to match all the rules inside that chain, in sequential order.

If one rule matches, the packet is processed following the specified action

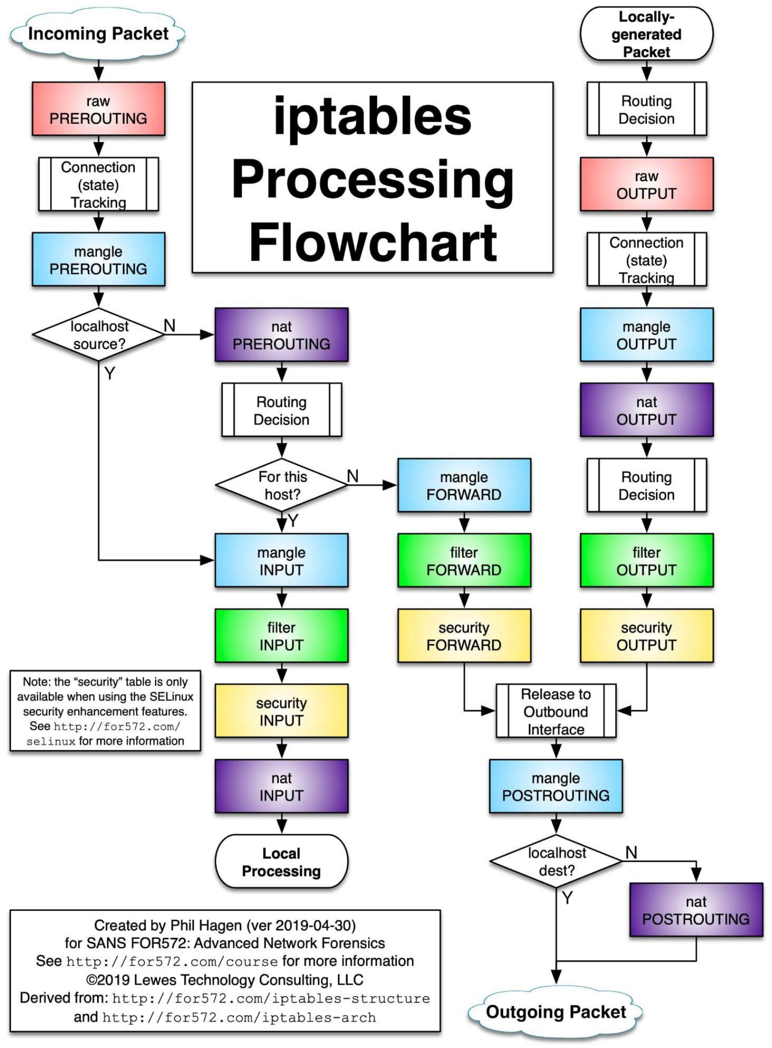
Packet flow inside iptables

For each packet that goes through the firewall, iptables tries to match rules on it. In the figure you can see the flow of packets inside the iptables stack.

We will only make use of tables **filter** and **nat**. They allow us to filter packets (accept or drop them) and to modify source or destination address and port of packets

legend:

table name
CHAIN NAME



Note: the "security" table is only available when using the SELinux security enhancement features. See `http://for572.com/selinux` for more information

Created by Phil Hagen (ver 2019-04-30)
for SANS FOR572: Advanced Network Forensics
See `http://for572.com/course` for more information
©2019 Lewes Technology Consulting, LLC
Derived from: `http://for572.com/iptables-structure`
and `http://for572.com/iptables-arch`

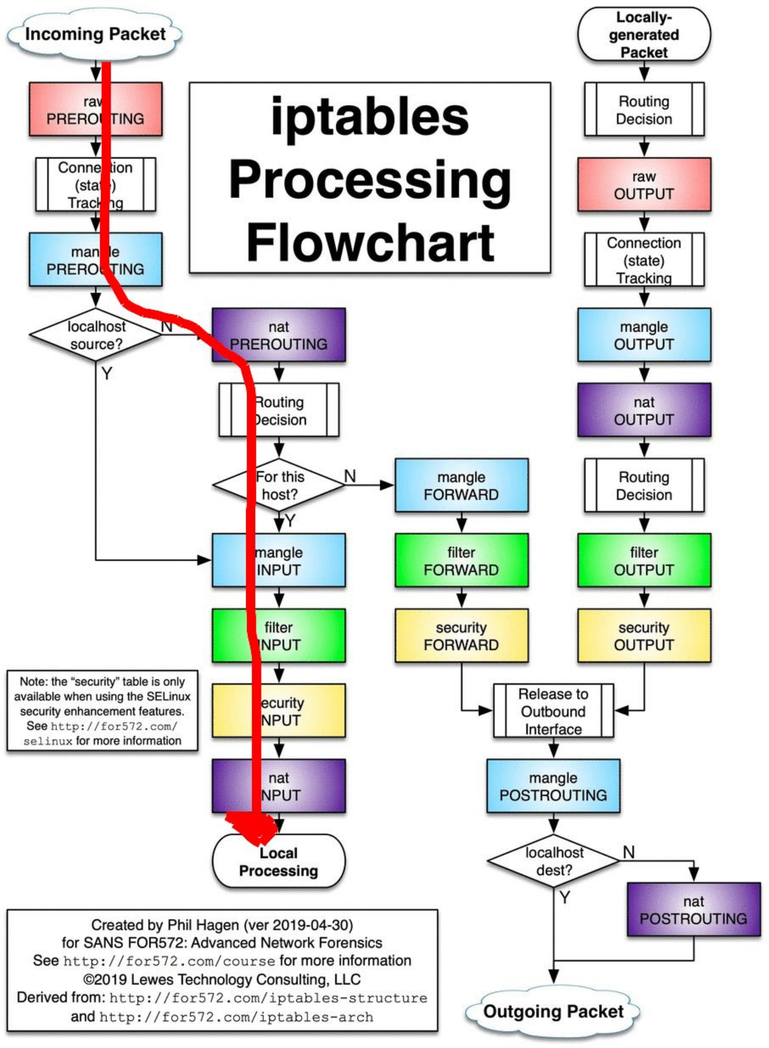
Packet flow inside iptables

Example:

Reception of packet destined to me

legend:

table name
CHAIN NAME



Note: the "security" table is only available when using the SELinux security enhancement features. See http://for572.com/selinux for more information

Created by Phil Hagen (ver 2019-04-30)
for SANS FOR572: Advanced Network Forensics
See http://for572.com/course for more information
©2019 Lewes Technology Consulting, LLC
Derived from: http://for572.com/iptables-structure
and http://for572.com/iptables-arch

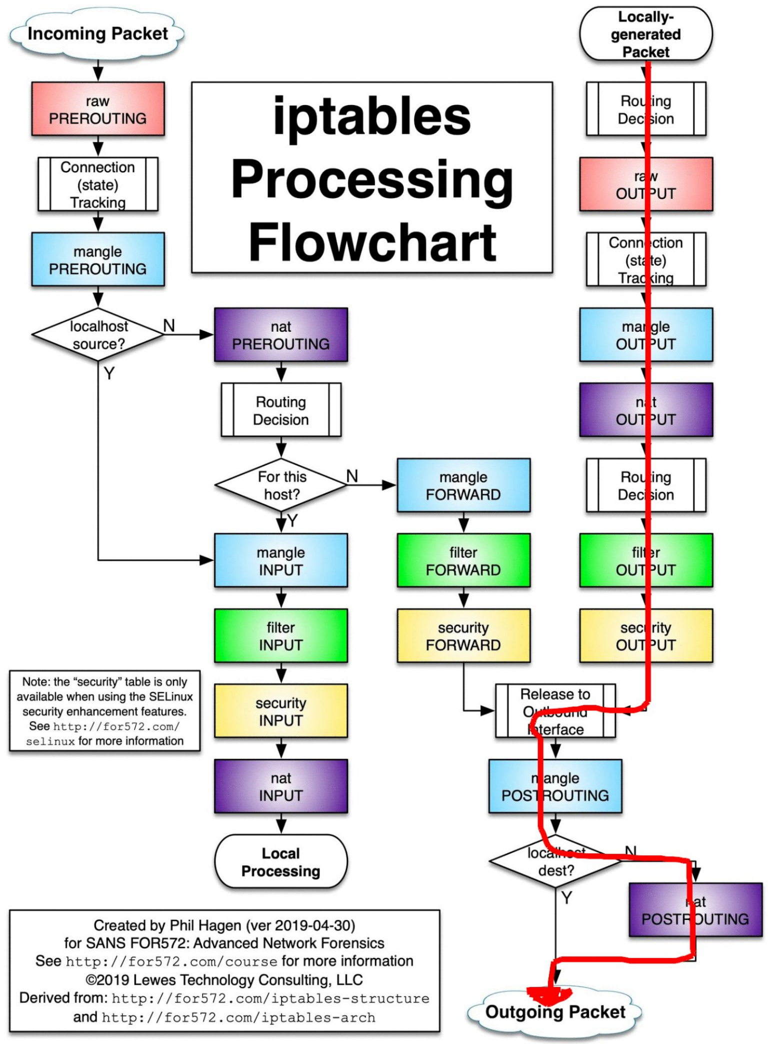
Packet flow inside iptables

Example:

Send a packet generated by me

legend:

table name
CHAIN NAME



Incoming Packet
raw PREROUTING
Connection (state) Tracking
mangle PREROUTING
localhost source?
N
Y
nat PREROUTING
Routing Decision
For this host?
N
Y
mangle FORWARD
filter FORWARD
security FORWARD
mangle INPUT
filter INPUT
security INPUT
nat INPUT
Local Processing
iptables Processing Flowchart
Locally-generated Packet
Routing Decision
raw OUTPUT
Connection (state) Tracking
mangle OUTPUT
nat OUTPUT
Routing Decision
filter OUTPUT
security OUTPUT
Release to Outbound Interface
mangle POSTROUTING
localhost dest?
N
Y
nat POSTROUTING
Outgoing Packet
Note: the "security" table is only available when using the SELinux security enhancement features. See http://for572.com/selinux for more information
Created by Phil Hagen (ver 2019-04-30) for SANS FOR572: Advanced Network Forensics
See http://for572.com/course for more information
©2019 Lewes Technology Consulting, LLC
Derived from: http://for572.com/iptables-structure and http://for572.com/iptables-arch

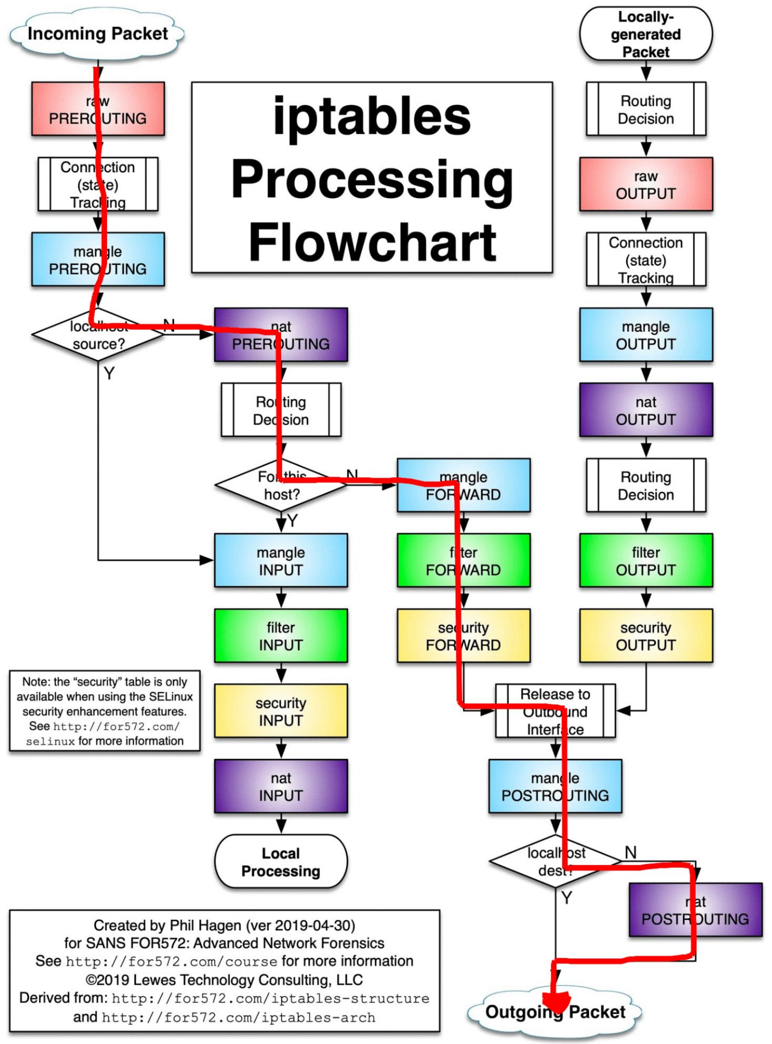
Packet flow inside iptables

Example:

Forward a packet generated by some host and directed to some other host (what usually a router does)

legend:

table name
CHAIN NAME



Note: the "security" table is only available when using the SELinux security enhancement features. See `http://for572.com/selinux` for more information

Created by Phil Hagen (ver 2019-04-30)
for SANS FOR572: Advanced Network Forensics
See `http://for572.com/course` for more information
©2019 Lewes Technology Consulting, LLC
Derived from: `http://for572.com/iptables-structure`
and `http://for572.com/iptables-arch`

iptables syntax

Append, delete, insert rules:

- iptables [-t table] {-A | D} chain rule-specification
- iptables [-t table] -D chain rulenum
- iptables [-t table] -I chain [rulenum] rule-specification

Flush, list rules:

- iptables [-t table] -F [chain [rulenum]]
- iptables [-t table] -{S | L} [chain [rulenum]] [options...]

Create, delete, rename chains and set default policy to a chain

- iptables [-t table] -N chain
- iptables [-t table] -X [chain]
- iptables [-t table] -E old-chain-name new-chain-name
- iptables [-t table] -P chain target

If -t is not specified it goes to the default “filter” table

Tables

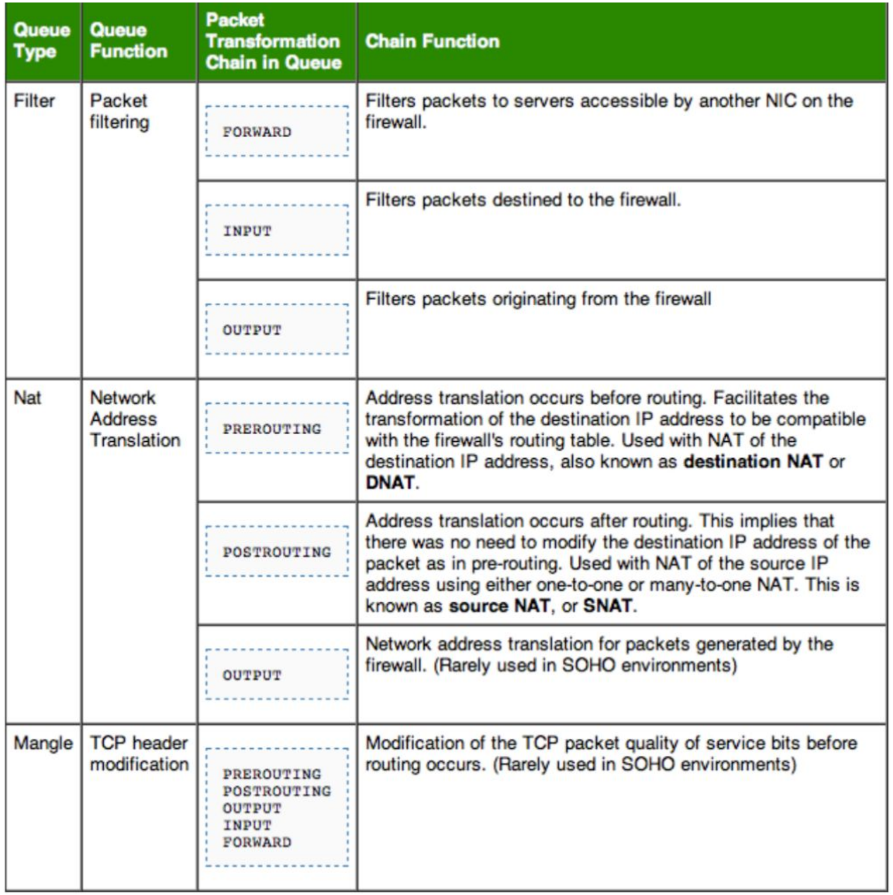
filter: This is the default table (if no -t option is passed . It contains the built-in chains **INPUT** (for packets destined to local sockets , **FORWARD** (for packets being routed through the box), and **OUTPUT** (for locally-generated packets)

nat: This table is consulted when a packet that creates a new connection is encountered. It consists of three built-ins: **PREROUTING** (for altering packets as soon as they come in), **OUTPUT** for altering locally-generated packets before routing), and **POSTROUTING** for altering packets as they are about to go out)

mangle: This table is used for specialized packet alteration. Until kernel 2.4.17 it had two built-in chains: **PREROUTING** (for altering incoming packets before routing) and **OUTPUT** (for altering locally-generated packets before routing). Since kernel 2.4.18, three other built-in chains are also supported: **INPUT** (for packets coming into the box itself), **FORWARD** (for altering packets being routed through the box), and **POSTROUTING** (for altering packets as they are about to go out)

raw: This table is used mainly for configuring exemptions from connection tracking in combination with the NOTRACK target. It registers at the netfilter hooks with higher priority and is thus called before ip_conntrack, or any other IP tables. It provides the following built-in chains: **PREROUTING** (for packets arriving via any network interface) **OUTPUT** (for packets generated by local processes)

Tables



Queue Type	Queue Function	Packet Transformation Chain in Queue	Chain Function
Filter	Packet filtering	`FORWARD`	Filters packets to servers accessible by another NIC on the firewall.
		`INPUT`	Filters packets destined to the firewall.
		`OUTPUT`	Filters packets originating from the firewall
Nat	Network Address Translation	`PREROUTING`	Address translation occurs before routing. Facilitates the transformation of the destination IP address to be compatible with the firewall's routing table. Used with NAT of the destination IP address, also known as **destination NAT** or **DNAT**.
		`POSTROUTING`	Address translation occurs after routing. This implies that there was no need to modify the destination IP address of the packet as in pre-routing. Used with NAT of the source IP address using either one-to-one or many-to-one NAT. This is known as **source NAT**, or **SNAT**.
		`OUTPUT`	Network address translation for packets generated by the firewall. (Rarely used in SOHO environments)
Mangle	TCP header modification	`PREROUTING` `POSTROUTING` `OUTPUT` `INPUT` `FORWARD`	Modification of the TCP packet quality of service bits before routing occurs. (Rarely used in SOHO environments)

Basic packet filtering - matching

The following parameters make up a match specification: (Iptables also support match extensions that provides many more match specification. More later on...)

- **[!] -p, --protocol protocol**: The protocol of the rule or of the packet to check. The specified protocol can be one of tcp, udp, udplite, icmp, esp, ah, sctp or all, or it can be a numeric value, representing one of these protocols or a different one. A protocol name from /etc/protocols is also allowed. The number zero is equivalent to all. The character "!" inverts the test
- **[!] -s, --source address[/mask]**: Source specification. Address can be either a network name, a hostname, a network IP address (with /mask), or a plain IP address. Hostnames will be resolved once only, before the rule is submitted to the kernel. Please note that specifying any name to be resolved with a remote query such as DNS is a really bad idea. The character "!" inverts the test
- **[!] -d, --destination address[/mask]**: Destination specification. See the description of the -s (source)
- **[!] -i, --in-interface name**: Name of an interface via which a packet was received (only for packets entering the INPUT, FORWARD and PREROUTING chains). When the "!" argument is used before the interface name, the sense is inverted. If the interface name ends in a "+", then any interface which begins with this name will match. If this option is omitted, any interface name will match.
- **[!] -o, --out-interface name**: Name of an interface via which a packet is going to be sent (for packets entering the FORWARD, OUTPUT and POSTROUTING chains). When the "!" argument is used before the interface name, the sense is inverted. If the interface name ends in a "+", then any interface which begins with this name will match. If this option is omitted, any interface name will match

Basic packet filtering - targets

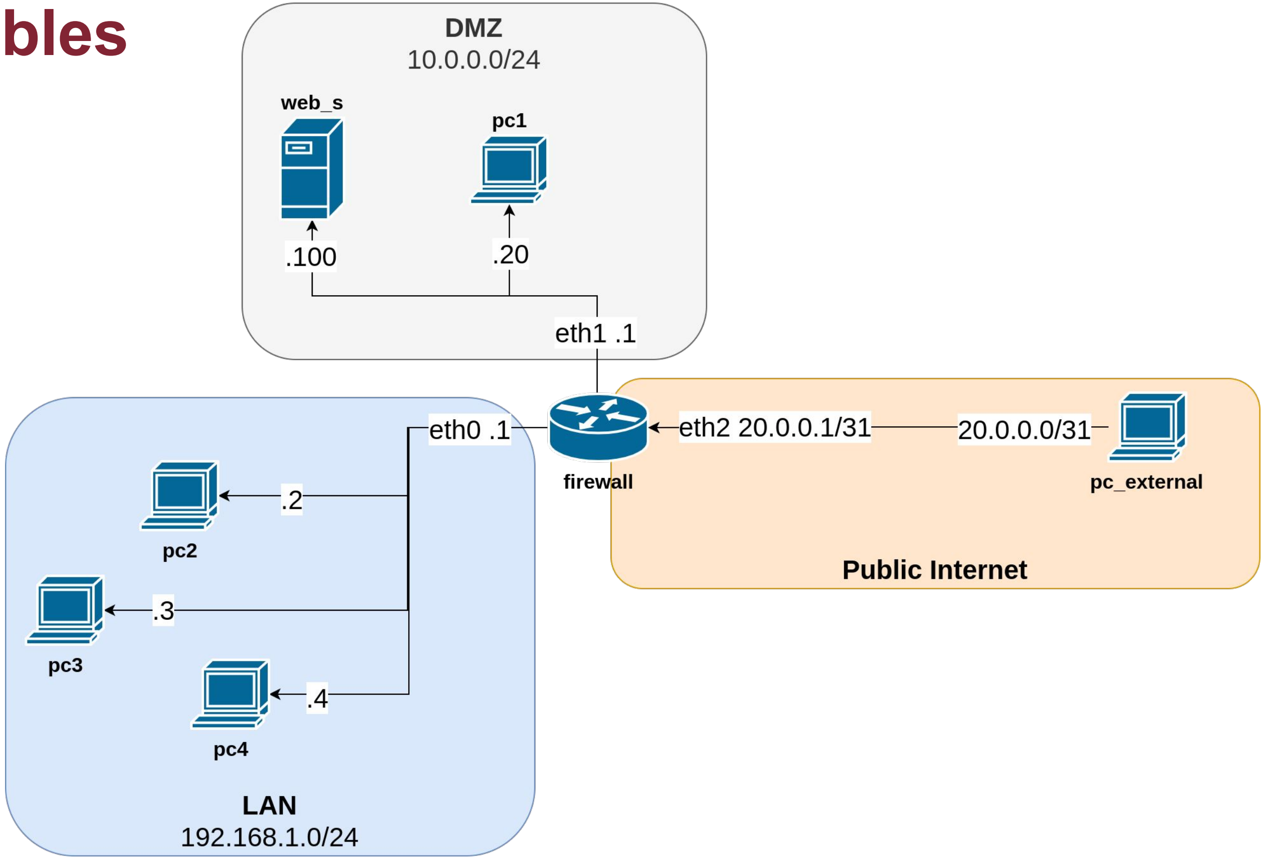
Targets are the actions that the firewall performs on packets matching a specification rule.

A target is specified with the option -j, and it can be the name of a user-defined chain or one of the special (standard) values:

- ACCEPT means to let the packet through (no other rules will be checked)
- DROP means to drop the packet silently
- REJECT means to drop the packet and reply to the sender with some icmp/tcp error
- RETURN means stop traversing this chain and resume at the next rule in the previous (calling) chain. If the end of a built-in chain is reached or a rule in a built-in chain with target RETURN is matched, the target specified by the chain policy determines the fate of the packet

More targets with target extensions.

lab_iptables



Filtering commands example

Note that in the following commands we are using the `filter` table. This is the default table if we not specify a `-t` option.

Clean any existing rule on filter table

```
iptables -F
```

Accept forwarding from LAN to everywhere

```
iptables -A FORWARD -s 192.168.1.0/24 -j ACCEPT
```

Accept forwarding from internet to DMZ

```
iptables -A FORWARD -i eth2 -d 10.0.0.0/24 -j ACCEPT
```

Accept forwarding from pc1 to LAN

```
iptables -A FORWARD -s 10.0.0.20 -d 192.168.1.0/24 -j ACCEPT
```

Default policy DROP for forwarding packets (target executed when no other rule matches)

```
iptables -P FORWARD DROP
```

Rule order

We want to drop input packets directed **to the firewall itself** coming from DMZ, except for pc1

```
iptables -A INPUT -s 10.0.0.0/24 -j DROP
iptables -A INPUT -s 10.0.0.20 -j ACCEPT
```

This **does not work!**

The second rule is appended to the end of the INPUT chain. Therefore the first rule is executed before the second one. When a packet arrives from 10.0.0.20 it matches the first rule and gets dropped.

*Rules are matched in **order**!*

Possible solutions:

- Invert the rule order
- Insert the second rule instead of appending it (`-I` instead of `-A`)

Protocol matching

Accept input DNS packets from LAN (DNS runs on udp port 53)

```
iptables -A INPUT -s 192.168.1.0/24 -p udp --dport 53 -j ACCEPT
```

Accept input HTTP packets from LAN (HTTP runs on tcp port 80)

```
iptables -A INPUT -s 192.168.1.0/24 -p tcp --dport 80 -j ACCEPT
```

Accept ICMP traffic (e.g. pings) from anywhere

```
iptables -A INPUT -p icmp -j ACCEPT
```

--dport matches the destination port
--sport matches the source port
--tcp-flags matches TCP header flags
--syn matches packets with the SYN flag set

Example: forward ssh traffic

Allow forwarding of SSH traffic from DMZ to internet

```
iptables -A FORWARD -s 10.0.0.0/24 -o eth2 -p tcp --dport 22 -j ACCEPT
iptables -A FORWARD -i eth2 -d 10.0.0.0/24 -p tcp --sport 22 -j ACCEPT
```

(We need **two** iptables rules, because we have to match the SSH packets going from DMZ to internet and the ones going from internet to DMZ)

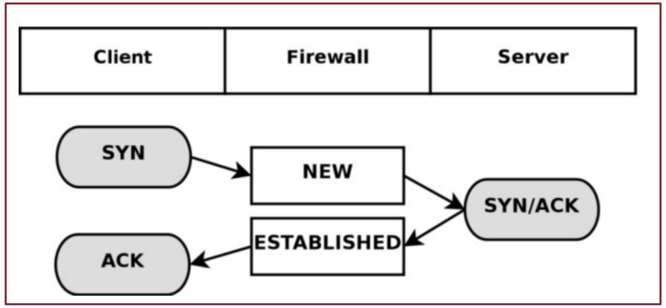
A similar way to do this is to allow all packets belonging to an existing connection:

```
iptables -A FORWARD -s 10.0.0.0/24 -o eth2 -p tcp --dport 22 -j ACCEPT
iptables -A FORWARD -m state --state RELATED,ESTABLISHED -j ACCEPT
```

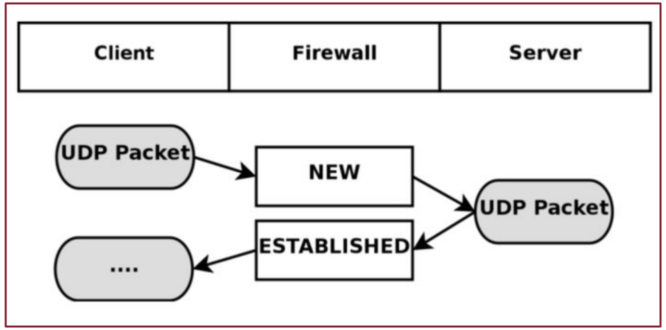
The first rule matches the first ssh packets sent by a host in DMZ. The second rule matches any packet belonging to the same connection (including the packets going from internet to DMZ). This second method is less strict than the first one.

Connection tracking

TCP connection opening



UDP “connection opening”



Any other packet of the same connection, whether is sent by the client or the server, is considered as ESTABLISHED.

Connection tracking

iptables is extensible, it gives you the possibility to load modules defining own matching rules. For connection tracking we use the `state` module

```
-m state [!] --state state
```

`state` is a comma separated list of connection states to match. Possible states:

- **INVALID** meaning that the packet could not be identified for some reason which includes running out of memory and ICMP errors which don't correspond to any known connection
- **ESTABLISHED** meaning that the packet is associated with a connection which has seen packets in both directions
- **NEW** meaning that the packet has started a new connection, or otherwise associated with a connection which has not seen packets in both directions
- **RELATED** meaning that the packet is starting a new connection, but is associated with an existing connection, such as an FTP data transfer, or an ICMP error

Network Address Translation - NAT

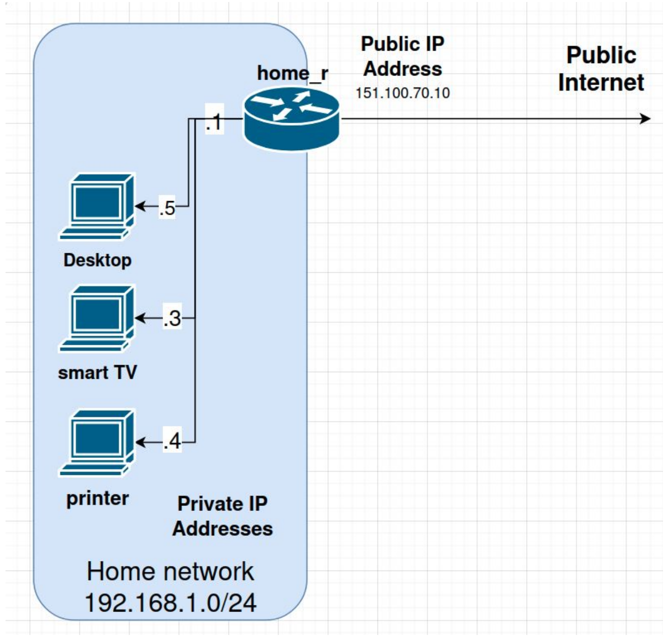
Throughout the course we talked about **private addresses** as addresses belonging to a LAN which are not publicly routable and addressable:

E.g. your router at home gives you the address 192.168.1.15, that address is private. It is not routable from the outside. It is not unique in the whole world.

- When you send packets to something on the internet, your home router (which has a **public** ip address) **substitutes** your **source ip** address **with its own**. When packets of the *same connection* come back to you, the router does the inverse process: substituting the destination ip address with 192.168.1.15 and forwards the packet to you.

This process, is commonly known as **NAT.**

Network Address Translation - NAT



Network Address Translation - NAT

NATs were invented to overcome the relatively small space of IPv4 addresses.

- Many hosts under the same “umbrella” of one single public IP address, since now IPv4 addresses are scarce and costly
- In IPv6 there is no need to worry about the quantity of addresses

NATs nowadays are used as “defense”: they avoid **exposing** a host to the public internet. The block any **new** connection **starting from outside** the LAN.

- *Observation*. If the attack is started from inside the LAN they don’t provide any protection
    - e.g. a *reverse shell*:
        1. an attacker may **install** a malicious program inside a PC inside a LAN.
        2. This program **connects** to a malicious remote host on the public internet (the NAT let this connection through as it is originated from the inside the LAN)
        3. The remote host and the PC are now directly connected: since a connection is *bidirectional*, the remote host can communicate directly with PC.

Source NAT with iptables

The NAT process can be specified in iptables as follows

```
    iptables -t nat -A POSTROUTING -o eth2 -j MASQUERADE
```

The previous command:

- Considers the `nat` table
- Considers packets **after** the routing decision has taken place (POSTROUTING) chain
- Considers all packets exiting from eth2 network interface
- Applies the target MASQUERADE, which implements the process of source and dest IP address substitution

We may also want to add the following rule:

```
    iptables -A FORWARD -i eth2 -m state --state NEW -j DROP
```

In this way we block any ***new*** connection **from the outside** of the internet **to our NAT** clients.

Test this in iptables_lab: if you launch a netcat listener with `-v` option on pc_external and you connect to it from a PC in the LAN, you can observe that the IP source address is the one of firewall

Destination NAT - DNAT

What we have seen in the previous slides is known as Source NAT or **SNAT** (it's called *source* since we are modifying the source address).

We can also do the Destination NAT or **DNAT**, where we change the destination IP and/or the port of a packet

Example: expose port 80 of firewall and redirect packets to web_s port 8080

```
iptables -t nat -A PREROUTING -i eth2 -p tcp --dport 80
    -j DNAT --to-destination 10.0.0.100:8080
```

The previous command:

- Considers packets before the routing decision has taken place (PREROUTING) chain
- Considers all packets entering from eth2 network interface
- Considers HTTP packets
- Applies the target DNAT redirecting packets to web_s port 8080.

- This is the exact process that occurs in your home router when you forward a port to some host
- This is useful in case you want to people to reach you on a privileged port (e.g. 80) but you want to run your server not as superuser

Basic iptables troubleshooting

Help! The firewall is dropping packets but I don’t know why

Use the following:

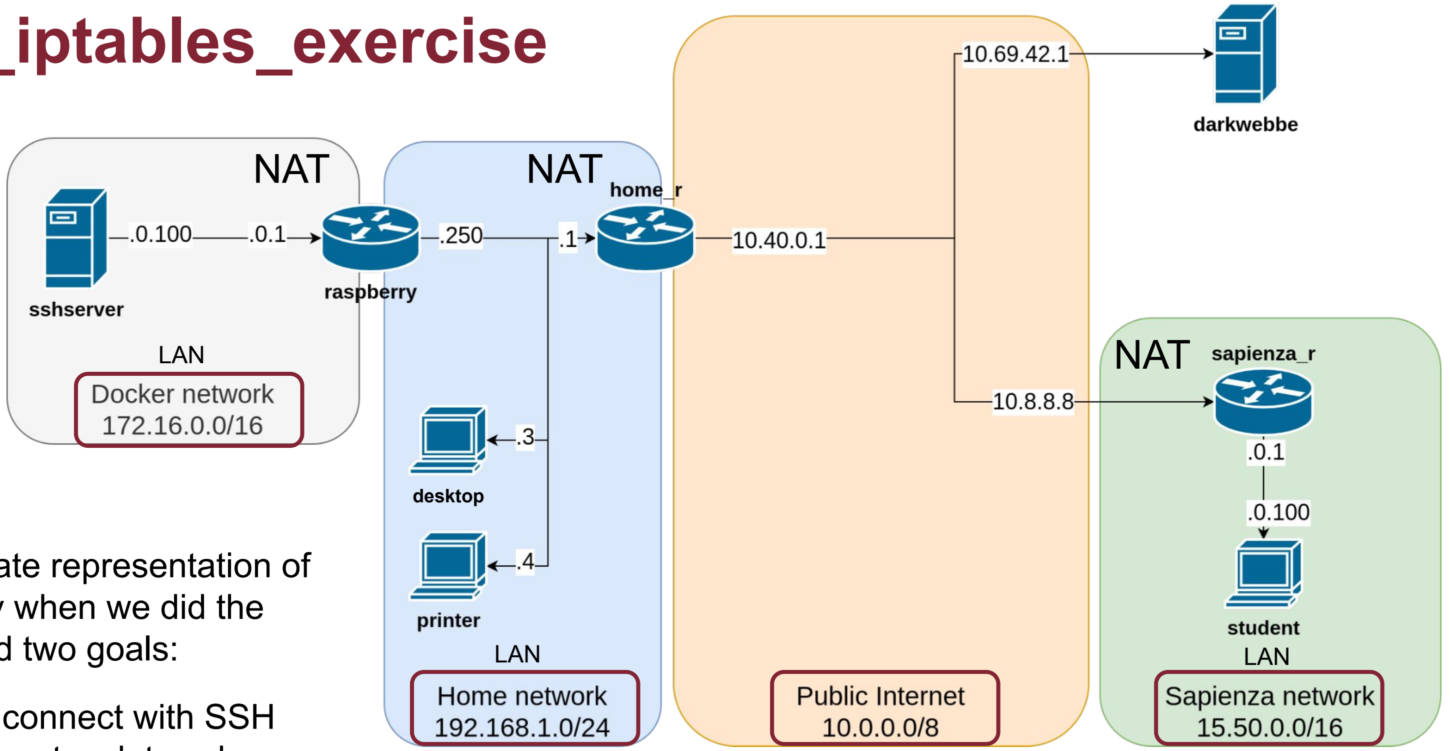
```
    iptables -t mytable -L -v -n
```

(where `mytable` is the table you want to inspect, e.g. filter or nat)

It shows you the rules inside the table as well as counters telling you how many packets and bytes matched some rule.

If you have some rule that is dropping traffic, but you don’t know which rule it is, you can keep track of the counters and identify that rule

lab_iptables_exercise



This is a semi-accurate representation of the network topology when we did the SSH laboratory. I had two goals:

1. Enable you to connect with SSH from Sapienza network to sshserver
2. Once inside sshserver, prevent you from connecting to other stuff

Networks Docker, Home and Sapienza are LANs **NATted** with private addresses:
→ <u>Connections **exiting** such LANs will have as **source** ip address the one of their gateway</u>

lab_iptables_exercise

In this lab you will configure the firewalls to perform some tasks acting only on **some specific nodes under your control**

- **Use the provided lab_iptables_exercise**
- Lab conf., networking and private addresses and NAT already done (in .startup file of `sapienza_r`, `home_r` and `raspberry`)

Tasks:

1. Acting *only* on `home_r` *and* `raspberry` add firewall rules to:
    - Enable SSH connection from `student` to `ssh_server`, so that `student` is able to login via ssh specifying the public address of **home_r**:
        - `ssh ssh_user@10.40.0.1`
2. Acting *only* on `raspberry` add firewall rules to block all connections originated from `sshserver` and...
    - Directed to `raspberry`
    - Directed to any of the nodes in my home network (eg. `printer`)
    - Directed to any of the nodes of the public internet

(To test if a node is reachable you can launch `nc ip_address 8080`)

If at the end all was done correctly:

1. `student` should be able to do: `ssh ssh_user@10.40.0.1`
2. Inside ssh, one should not be able to connect to raspberry, printer, desktop or darkwebbe