Hardening Network

Fortificación de S.O.
Master en Seguridad Informática. 2023/24
Universidade da Coruña
Universidade de Vigo

Antonio Yáñez Izquierdo
José Rodríguez Pereira

Contents I

Contents II

Introduction: Network Configuration in Debian based Linux systems

Introduction: Network Configuration in Debian based Linux systems
→Basic network configuration

basic IP v4 configuration

- to properly configure a machine using ipv4 we have to configure
  - the machine name
  - the Network Interface Cards
  - the routes
  - the dns (if using it)

basic NIC configuration

- The basic things we have to configure for a Network Interface Card are
  - its ip address
  - its netmask (number of bits in its ip address that correspond to network address)
  - its broadcast address

ways to configure the network

- there are two ways to configure the network
  - **manual configuration:** we configure manually each of the paramaters, either directly using the comand line or through the boot scripts
  - **using dhcp:** the network interface card *asks for its configuration* to a machine in the network (the *dhcp server*). This can be done directly through the comand line or using the boot scripts
- most systems have a graphic utility to configure the network, which can be used to configure either manually or via *dhcp*. We won't deal with those utilities.

ifconfig

- the comands **ifconfig** an **ip** configure network interfaces,
- they are usually located at **/sbin**
- they can configure interfaces both manually or using dhcp
- `ifconfig -a` or `ap addr show` show the actual configuration of the Network Interface Cards
- **ifconfig** is being superseded by **ip** to the point that some linux distros do not even install it by default

configuring the dns

- the configuration of the *dns* resides on the file `/etc/resolv.conf`
- this file has the options to the *resolver* configuration. The most common options are
  - **nameserver** to specify the address of a domain name server, up to 3 can be defined
  - **domain** (optional) to sepecify the local domain. Short names are supposed to be from this domain
- example of `/etc/resolv.conf` file

```
domain dc.if.udc.es.
nameserver 193.144.51.10
nameserver 192.144.48.30
```

the /etc/hosts file

- this file contains the locally defined ip addresses of hosts
- its format is

  `ip_address  host_name  aliases`
- example of `/etc/hosts`

```
127.0.0.1       localhost
192.168.1.99    abyecto.dc.fi.udc.es    abyecto
```

the /etc/nsswitch.conf file

- this file is used to determine the sources from where to obtain name-service information of several categories: hosts, users, mail aliases ...
- it also specifies the order in which this sources of information should be queried
- in the following example, the hosts ips are first searched for in the local files, then the dns is queried

```
passwd:         compat
group:          compat
shadow:         compat

hosts:          files dns
networks:       files
```

Introduction: Network Configuration in Debian based Linux systems
→Naming network devices

Naming Network Interfaces

- linux distros have followed several namimg strategies for namimg NICS
  1. linux used to name their NICs eth0, eth1 . . . and the order was defined by which one got detected first
     - this makes order dependent on module loading, and changing one NIC for other could change all the names
  2. the names *eth0, eth1, eth2 . . .* are asigned to the interfaces **THE FIRST TIME** the kernel recognices them. This is stored in the file `/etc/udev/rules.d/70-persistent-net.rules`, where it can be changed if necessary.
  3. they get the names like em*N*, emp*N*s*M*, ens*N*, p*N*p*M*. This new name scheme does not make names dependent on the type of card, its mac or when it is detected; the names are generated depending on how (where) they are connected to the system which makes it easier to substitute interfaces
- debian and its derivatives use ways *2* or *3* to name the interface

Example of70-persistent-net.rules file

```
abyecto:/home/antonio# cat /etc/udev/rules.d/70-persistent-net.rules
# This file was automatically generated by the /lib/udev/write_net_rules
# program, run by the persistent-net-generator.rules rules file.
#
# You can modify it, as long as you keep each rule on a single
# line, and change only the value of the NAME= key.

# PCI device 0x11ab:0x4363 (sky2)
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="00:24:be:40:5c:
ATTR{type}=="1", KERNEL=="eth*", NAME="eth0"

# PCI device 0x8086:0x4232 (iwlagn)
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="00:24:d6:0e:ae:
ATTR{type}=="1", KERNEL=="wlan*", NAME="wlan0"
abyecto:/home/antonio#
```

Introduction: Network Configuration in Debian based Linux systems
→NIC configuration in debian linux and its derivatives

NIC configuration in debian linux and its derivatives

- **dhclient interface_name** configures the card *interface_name* using dhcp.
- **ifconfig interface_name inet address addr netmask netmk broadcast bcast** configures the card *interface_name* with address *addr*, netmask *netmk* and broadcast address *bcast*.
- **ip addr add address dev interface_name** configures the card *interface_name* with address *addr*

```
#ifconfig eth0 inet 192.168.1.100 netmask 255.255.255.0 broadcast 192.168.1.255
#ip addr del 192.168.2.100 dev p2p1
```

- **ifconfig interface_name up** brings the interface up as does **ip link set interface_name up**

NIC configuration in debian linux at boot time

- if we want to get the interfaces automatically configured at boot time (via `/etc/init.d/networking` or `systemctl`)
- debian systems and derivatives will look for the file `/etc/network/interfaces` (see `interfaces` man page)
- `/etc/hostname` Contains the name of the system (either the fully qualified domain name or just the nodename)

NIC configuration in debian linux and derivatives at boot time

- Sample `/etc/network/interfaces` with just one NIC manually configured

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
#allow-hotplug eth0
auto eth0
iface eth0 inet static
address 192.168.1.99
netmask 255.255.255.0
network 192.168.1.0
broadcast 192.168.1.255
gateway 192.168.1.1
```

NIC configuration in debian linux and derivatives at boot time

- Sample `/etc/network/interfaces` with just two NICs

```
root@abyecto:~# cat /etc/network/interfaces
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo eth0 eth1
iface lo inet loopback

# The primary network interface
allow-hotplug eth0
iface eth0 inet dhcp
# internal network
allow-hotplug eth1
iface eth1 inet static
address 192.168.1.100
netmask 255.255.255.0
network 192.168.1.0
breadcast 192.168.1.255
```

Introduction: Network Configuration in Debian based Linux systems
→Interaction with Network Manager

Interacción con Network Manager

- *Network Manager* is a package that gets instaled on most desktop linuxes
- Consists of a *daemon* executing in the background `root` and a *font-end* dependant on the desktop environment
- *Network Manager* wpould try to manage all NICs non declared on the system's configuration files
- To see the devices managed by*Network Manager*

  `#nmcli dev status`
- *Network Manager*'s configurations resides on `/etc/NetworkManager/NetworkManager.conf`

Interfaces managed by Network Manager

- If we want an interface NOT MANAGED by *Network Manager* we must
  - get the interface configured at `/etc/network/interfaces`
  - have the following lines included in `/etc/NetworkManager/NetworkManager.conf`

```
[main]
plugins=ifupdown
[ifupdown]
managed=false
```

- interfaces managed by the Network Manager can be also configured by the `nmcli` command

Network Manager

- If we want NetworkManager to be temporarily stopped *Network Manager* one of these commands (depending on the distribution)

```
# service NetworkManager stop
# /etc/init.d/network-mmanager stop
# systemctl stop NetworkManager.service
```

- If we want it no to be started at boot time

```
# chkconfig NetworkManager off
# update-rc.d network-manager remove
# insserv -r network-manager
# systemctl disable  NetworkManager.service
```

Introduction: Network Configuration in Debian based Linux systems
→Network interface aliasing

interface aliasing

- By interface aliasing we refer to the act of giving a Network Interface Card more than one IP address.
  - the simplest way is to configure these alias as we would do with a *non-aliased* interface but using the names eth0:0, eth0:1..., eth1:0....
    To get it configured at boot time we just add an entry for it in the file `/etc/network/interfaces` as we would do with a *non-aliased* interface.

```
# ifconfig eth0:0 192.168.1.11 up
# ip addr add 192.168.1.45 dev eth0
```

interface aliasing

- As the **ip** command allows an interface to have multiple addresses we can configure these alias as we would do with a *non-aliased* interface using the *ip addr* command.
  - We simply add internet addresses to the interface

```
# ip addr add 192.168.2.100 dev p2p1
# ip addr add 192.168.29.18 dev p2p1
```

- if we want this to get configured at boot time we can add addresses to the interfaces at `/etc/network/interfaces` file

example of /etc/network/interface

```
# This file describes the network interfaces available on your s
# and how to activate them. For more information, see interfaces

# The loopback network interface
auto lo
iface lo inet loopback
auto enp0s3
iface enp0s3  inet   dhcp
auto enp0s8
iface enp0s8   inet  static
        address 192.168.10.102/24
iface enp0s8   inet  static
        address 192.168.11.102/24
iface enp0s8   inet  static
        address 192.168.12.102/24
```

interface aliasing: old way

- we cal also define the aliased interfaces as *interface_name:number*. The example above would look like this

```
.....
auto enp0s8
iface enp0s8   inet  static
        address 192.168.10.102
        netmask 255.255.255.0
auto enp0s8:1
iface enp0s8:1   inet  static
        address 192.168.11.102
        net,ask 255.255.255.0
auto enp0s8:2
iface enp0s8:2   inet  static
        address 192.168.12.102
        netmask 255.255.255.0
```

- aliases defined this way are visible with the *old* `ifconfig` command

inetd

inetd

- **inetd** is called the internet superserver.
- Some internet services listen directly to their corresponding port, others are started by **inetd**
- When a conexion request arrives on a designated port, **inetd** starts the appropiated server program
- This allows for server programs to run only when needed, thus saving resources on the system
- Two files control the working of **inetd**
  - `/etc/services`
  - `/etc/inetd.conf`

/etc/services

- /etc/inet/services on some systems
- this file has a mapping between the port numbers and protocol to the services names. Info can be found in the services man page. A fragment from an acual /etc/services is shown

```
ftp             21/tcp
fsp             21/udp          fspd
ssh             22/tcp                          # SSH Remote Login Protocol
ssh             22/udp
telnet          23/tcp
smtp            25/tcp          mail
time            37/tcp          timserver
time            37/udp          timserver
rlp             39/udp          resource        # resource location
nameserver      42/tcp          name            # IEN 116
whois           43/tcp          nicname
```

/etc/inetd.conf

- This file associates the service name to the program actually providing the service
- The format for one line of this file is

```
service_name  socket_type  protocol  wait/nowait  user.group  program  args
```

/etc/inetd.conf

- As lines started with the # are treated as comments, we can disable one service, by simply comenting out the corresponding line
- Example of the `telnetd` service disabled

```
#telnet   stream  tcp     nowait  root    /usr/sbin/in.telnetd    in.telnetd
```

- debian linux does not include `inetd`, it can be installed as package `openbsd-inetd` (usually as a dependence of other network packages)

inetd in fedora linux

- Fedora linux, (as do some distrbutions of linux), does not include `inetd`. It includes `xinetd` a inetd replacement
- However, it is no necessary to use `xinetd` to use such services such as *telnetd* or *ftpd*
- For example, should have we installed `pure-ftpd` as the ftp server we can enable that ftp service by doing

```
# systemctl enable pure-ftpd
# systemctl start pure-ftpd
```

Access control: tcpwrappers

tcpwrappers

- An aditional layer can be placed between `inetd` and the server program to perform access control based on host name, network address or ident queries
- This layer is usally called `tcpwrappers` or, by the name of the program, `tcpd`.
  - the program `tcpd` gets called by `inetd` and receives the server to start as a parameter
  - `tcpd` checks its configuration files to see if the access must be granted or denied
  - in case the access is granted `tcpd` starts the server program supplied as parameter
- the corresponding line for ths `telnetd` server using *tcpwrappers* would look like this

```
telnet   stream  tcp  nowait  telnetd /usr/sbin/tcpd  /usr/sbin/in.telnetd
```

- This allows to have access control at an application lever for applications that do not provide it

tcpwrappers

- the configuration for the *tcpwrappers* resides in the files `/etc/hosts.allow` and `/etc/hosts.deny`
- the manual page `hosts_access` documents the use of these files
  - Access will be granted when a (daemon,client) pair matches an entry in the `/etc/hosts.allow` file.
  - Otherwise, access will be denied when a (daemon,client) pair matches an entry in the `/etc/hosts.deny` file.
  - Otherwise, access will be granted.

tcpwrappers

- `xinetd` can also implement this access control
- programs that have been compiled with *libtcpwrappers* (some times called *libtcpd* or *libwrap*) support access control on their own (either called directly or through *inet*) and need not be called through *tcpd*
- Some modern versions of the *tcpwrappers* combo use a *'simplified'* format of the `/etc/hosts.allow` and `/etc/hosts.deny` files: only the `/etc/hosts.allow` is necessary,. As of version 12, debian still uses both files.
- the operator EXCEPT can be used to define more precisely a set of connections.

tcpwrappers: sample hosts.allow and hosts.deny entries

- the following configuration woul allow ssh conectiond from every machine, and ftp connections only from network 192.168.2 and domain example.com (we assume that in.ftpd is the ftp server program)

```
# cat /etc/hosts.allow
in.ftpd : 192.168.2.*
in.ftpd : .example.com
sshd : ALL
# cat /etc/hosts.deny
in.ftpd : ALL
```

Access control: Packet Filtering

Packet filtering

- A packet filter is a program the checks the headers of each network packet that reaches to it, and upon inspecting it, decides to perform an action such as rejecting it, drop it or accept it
- In linux we have such packet filter in the kernel as part of the packet managing infrastructure (Netfilter).
- Configuration is lost when rebooting the machine, so it must be included in some of the initiation scripts
- Netfilter contains different tables for the different functions it supports, being *filter* the table for packet filtering

Packet filtering

- the filter table operates on *chains*
- each *chain* has a set of rules that operate on the packets belonging to that chain
- rules are checked in order. When the packed matches one rule that action is executed and no more rules are checked for that packet
- should the packet not match any of the rules, the default action for that *chain* is taken

Access control: Packet Filtering
→linux packet filtering: iptables

Packet filtering

- we can define as many *chains* as we'd like to
- the system has three predefined *chains*: INPUT, OUTPUT and FORWARD

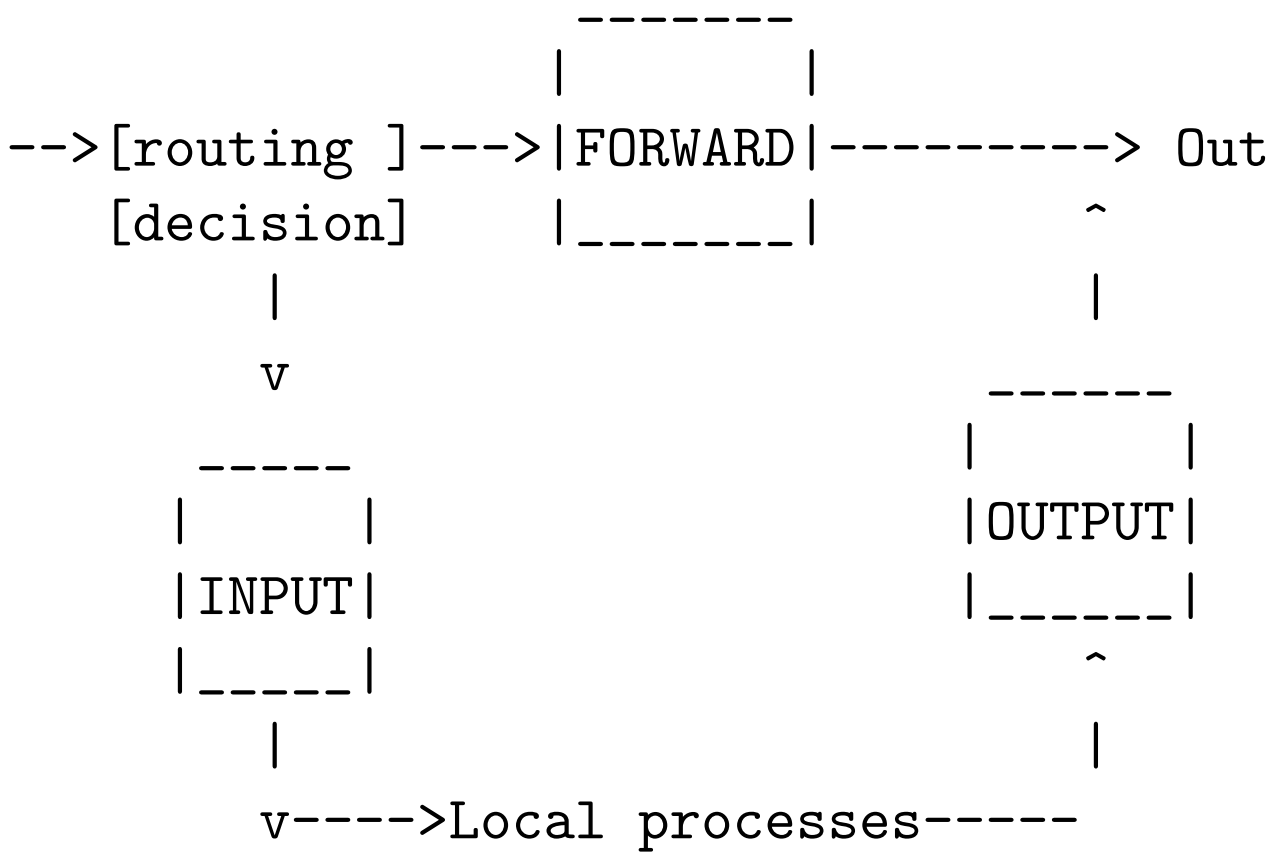
INPUT — packets that intend to go to one or our system's processes

OUTPUT — packets that originate in one of our machine processes and go out

FORWARD — packets that arrive to our machine and go to another. This chain is of concern only to machines acting as routers

Packet filtering

- packets and chains.



Iptables: chain manipulation

- we use the program *iptables* to modify the netfilter table
- we can use it to manipulate the *chains*
  - *chain* creation: iptables -N *chain_name*.
  - *chain* deletion: iptables -X *chain_name*.
  - changing default *chain* policy:
    iptables -P *chain_name action*, where action can be
    - DROP packet is discarded
    - ACCEPT packet is accepted
    - REJECT packet is discarded and an ICMP is sent to the sender

Iptables: chain manipulation

- list rules on a chain: iptables -L *chain_name*.
- selete all the rules ona a chain (flush): iptables -F *cadena*.
- here we have an example

```
root@hardenin:/home/antonio# iptables -L INPUT
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere
ACCEPT     icmp --  anywhere             anywhere             icmp any
DROP       tcp  --  anywhere             anywhere             tcp flags:FIN,SYN,RST,PSH,ACK,URG/FIN,SYN,RS
DROP       tcp  --  anywhere             anywhere             tcp flags:FIN,SYN,RST,PSH,ACK,URG/NONE
ACCEPT     all  --  anywhere             anywhere             state RELATED,ESTABLISHED
root@hardenin:/home/antonio# iptables -F INPUT
root@hardenin:/home/antonio# iptables -L INPUT
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
root@hardenin:/home/antonio#
```

Iptables: rule manipulation

- we can manipulate rules with the `iptables` command. Note that as rules are numbered operations such as delete or insert use the rules number (we gan get the actual numbers with the `--list-numbers` option when invoking `iptables -L`)
  - Add a rule to a chain: `iptables -A` *chain rulespec*
  - Delete a rule from a chain: `iptables -D` *chain rulenumber*.
  - Inserting a rule: `iptables -I` *chain number rulespec*.
  - Replacing a rule: `iptables -R` *chain number rulespec*.

Iptables: rules specification

- rules are formed by two components
  - packet selection, i.e. specify the packets on which the rule operates
  - action to take, i.e. what to do on those packets
- so, a rule specification (`rulespec` in the previous syntax), has the following form

```
packet_selection -j ACTION
```

Iptables: packet selection

- the first part of specifying a rule is the *packet selection*
- we can select a packet by
- protocol: `-p` *protocol* (*p*rotocol can be `tcp`, `udp`, `icmp` or `all`)
- source port: `--sport` *port*
- destination port: `--dport` *port*
- the following example selects a udp packet comming from port 156

```
-p udp --sport 156
```

Iptables: packet selection

- source address: `-s` *address/mask*
- destination address: `-d` *address/mask*
- input interface: `-i` *iface_name*
- output interface: `-o` *iface_name*
- we use `!` to deny, for example `-i ! eth1` means any input interface except eth1

Iptables: packet selection

- we can select packet fragmets (not the first fragment) with `-f`
- we can identify connecting packets `--syn`
- for icmp we can specify the type with `--icmp-type` *type*, example `-p icmp --icmp-type ping`
- with can check if the state matches some states, example `-m state --state STABLISHED,RELATED`

Iptables: Actions

- We specify what is to be done with the packet with `-j` *action*. *Action can be one of the following:*

DROP *the packet is dropped (this would be seen as* no response*)*

REJECT *the packet is rejected (this would be seen as* connection refused

ACCEPT *the packet is accepted*

LOG *a log entry is generated for this packed. This action does not end the rule checking for the packet*

Iptables: Example

- allowing all traffic in the loopback interface

```
# iptables -I INPUT 1 -i lo -j ACCEPT
```

- allowing stablished connections

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

- allowing incomming connections to web server (port 80) from anywhere

```
# iptables -A INPUT -p tcp --dport 80 -j ACCEPT
```

- allowing (and logging) ssh coonections to interface eth1 comming from network 193.144.51.0

```
# iptables -A INPUT -p tcp --dport 23 -i eth1 -s 193.144.51.0/24 -j LOG
# iptables -A INPUT -p tcp --dport 23 -i eth1 -s 193.144.51.0/24 -j ACCEPT
```

- rejecting pings.

```
# iptables -A INPUT -p icmp --icmp-type ping -j DROP
```

- allow stablished connections in interface eth0

```
# iptables -A INPUT -i eth0 -p tcp  -m state --state ESTABLISHED -j ACCEPT
```

- establishing default policy in input chain to DROP

```
# iptables -P INPUT DROP
```

Iptables: Saving and restoring configuration

- we can save the current configuration of the filter table with `iptables-save`
  - it writes the iptables configuration (list of rules) to the standard output
  - should we want it in a file we can redirect it to a file with `iptables-save > file_name`
- we can restore the configuration saved previously in a file (in the format `iptables-save` does) with `iptables-restore`
  - *file_name* has a previously saved configuration we could restore it with `iptables-restore file_name`

Access control: Packet Filtering
→linux packet filtering: nftables

nftables

- nftables is the modern Linux kernel packet classification framework
- available from Linux kernels 3.13
- rulesets can be arranged in treelike structure thus reducing the time to inspect each packet (iptables rules were secuential)
- accesible through the *nft* command
- we can still use the old *iptables* command to access the packet filter in the kernel.

andvantages of nftables

- Faster packet classification
- Simplified dual stack IPv4/IPv6 administration
- Nicer and more compact syntax
- Better dynamic ruleset updates

nftables: differences with iptables

- syntax is different
- *nftables* does not have predefined tables or chains
- a single rule of nftables can take more than one action: a rule consists of zero or more expressions and one or more statements. Expressions are evaluated left to right (AND logic, if one expression is matched we continue to the next). If the packet matches the last expression, then it has matched all the expressions and the statements are executed on it. As with the expressions, the statements are executed in order (left to right)
- support to new protocols could be added with user level software (intead of requiring a kernel upgrade)
- there exist a command *iptables-translate* that translates *iptables* rules to the equivalent *nft*

nftables: families

- rules defined *nft* get lost at rebooting
- if we want some rules to be permanent we include them at `/etc/nftables.conf` and will be loaded when the service is reinitiated
- to see our set of rules `nft list ruleset`
- rules are stored in chains, which in turn are stored in tables.
- there are not predefined tables or chains
- the number of tables and their names are used defined

nftables: families

- each table has only one family of addresses and applies only to addreses in that family
- families are

ip	ipv4 (*old* iptables command)
ip6	ipv6 (*old* ip6tables command)
inet	ipv4 and ipv6
bridge	bridge (*old* brtables command)
arp	arp (*old* arptables command)

- by default, the family *ip* is assumed

nftables: tables

- we can create a table with `'nft add table [family] table_name'` (if family is omited *ip* is assumed)

```
# nft add table Filtrado
```

- we can delete a table with `'nft delete table [family] table_name'`. *family* need only be specified if theres a table with the same name in different families. This would delete table *Filtrado* created before

```
# nft delete table Filtrado
```

- we can see the tables with `'nft list tables'`
- to flush a table we use `'nft flush table [family] table_name'`. Again *family* need only be specified if theres a table with the same name in different families.

nftables: chains

- chains store the rules that we're going to define
- there are no predefined chains (as is the case with *iptables*)
- there are two types of chains
  - Normal chains: can be used as targets for jumps
  - Base chains: a base chain is one that is registered into one of the Netfilter Hooks (see figure on the packets through the TCP/IP stack)
- to create a chain we use `'nft add chain [family] table_name chain_name'` For example, to create a chain in table *Filtrado* we would do

```
# nft add chain Filtrado cadenaPrimera
```

- to delete a chain we use `'nft delete chain [family] table_name chain_name'`
- we can see the chains we created with `'nft list chains'`

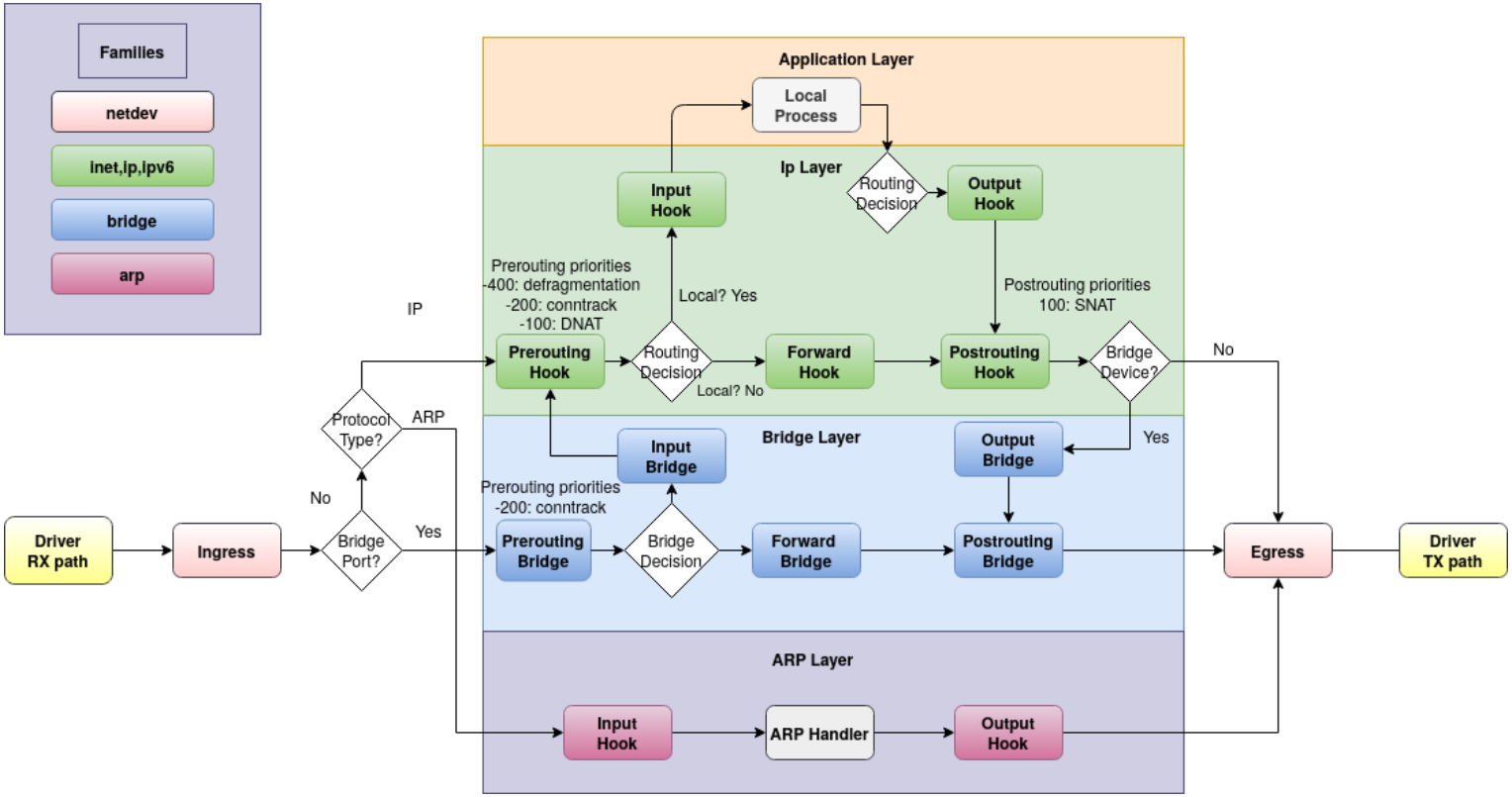
nftables: base chains

- to create a base chain we use `'nft add chain [family] table chain_name {type what_type hook what_hook priority prio; policy what_policy;}'`
- *type* can be either *filter*, *route* or *nat*
- *hook* dependes on the family. The *hooks* available are

ip/ip6/inet	prerouting, input, forward, output, postrouting
arp	input, output
bridge	prerouting, input, forward, output, postrouting

- *priority* is an integer. Chains with lower values are processed first (negative values can be used)
- *policy* can be one of `accept` or `drop`

nftables: netfilter hooks



nftables: base chains

- if we don't specifie *type what_type ...* into curly brackets a normal chain will be created instead of a base chain
- As the shell processes certain characteres (curly braces, the semicolon...), to create a base chain, we need to escape the characters or quote everything. Example

```
# nft add chain Filtrado cadenaEntrada '{ type filter hook input priority 1; policy accept;}'
```

- only base chains see packets through its *hook*; non base chains see backets when they are jumped to
- we can delete chains with `'nft delete chain [family] table chain_name'`, provided that the chain is empty.
  - if necessary, we may need to flush the chain first with `'nft flush chain [family] table chain_name'`

nftables: base chains

- When you create a chain, the priority specifies the order in which chains with the same hook value traverse
- For families ip,ip6 and inet we can specify the priority as an integer or use one of the predefined named priorities

name	**value**
raw	-300
mangle	-150
dstnat	-100
filter	0
security	50
srcnat	100

nftables: rules

- a table refers to a container of chains. A chain within a table refers to a container of rules. A rule refers to an action to be configured within a chain.
- to add a rule we use

  `nft add rule [family] table_name chain_name <matches> <statements>`

  - *matches* allow us tho select the packest to which we want the *statements applied*
  - *statements* the action (or actions we take on those packets)

nftables: rules

- if we add a rule with
  `nft add rule ...`
  that rule would get added at the end of the ruleset. With
  `nft insert rule [family] table_name chain_name <matches> <sttments>`
  the rule would get added at the begining of the rule set.

nftables: rules

- we can specify the position in which we want to add a rule

  `nft add rule [family] table_name chain_name position <handle> <matches> <statements>`

  adds a rule after the one with handle `handle`
- we can specify the position in which we want to add a rule

  `nft insert rule [family] table_name chain_name position <handle> <matches> <statements>`

  adds a rule before the one with handle `handle`
- we can delete a rule with

  `nft delete rule [family] table_name chain_name handle <handle>`. Examples

```
#nft add rule Filtrado cadenaEntrada position 6 tcp dport 22 drop
#nft insert rule Filtrado cadenaEntrada position 9 iifname ‘‘eth0’’ accep
#nft delete rule Filtrado cadenaEntrada handle 7
```

- to see the handles of the rules we use `nft -a` when listing

```
#nft -a list ruleset
#nft -a list chain Filtrado cadenaEntrada
```

nftables: rules matches

- most usual matches (**ip**)
  - **ip protocol prot**. *prot* can be `icmp`, `esp`, `ah`, `comp`, `udp`, `udplite`, `tcp`, `dccp`, `sctp`, a set as in {`tcp`, `udp`} or `!=`. Examples:

    ```
    ip protocol tcp
    ip protocol != {tcp, icmp}
    ```

  - **ip saddr|daddr addr**. *addr* can be an (source or destination) address, a set or range of addresses or != Examples:

    ```
    ip saddr != 192.168.2.0/24
    ip saddr 192.168.3.1 ip daddr 192.168.3.100
    ip saddr != 192.168.0.1-192.168.0.100
    ip daddr  { 192.168.2.1, 192.168.2.2, 192.168.2.3 }
    ```

nftables: rules matches

- most usual matches (**tcp udp**)
  - **tcp sport|dport port**. (source o destination) *port* can be a port number, name, a set, a range or != Examples:

    ```
    tcp dport {telnet, http, https }
    tcp sport != 33-45
    ```

  - **udp sport|dport port**. (source o destination) *port* can be a port number, name, a set, a range or !=

nftables: rules matches

- most usual matches (**icmp**)
  - **icmp type typ** *typ* can be any of `echo-reply`, `destination-unreachable`, `source-quench`, `redirect`, `echo-request`, `time-exceeded`, `parameter-problem`, `timestamp-request`, `timestamp-reply`, `info-request`, `info-reply`, `address-mask-request`, `address-mask-reply`, `router-advertisement`, `router-solicitation`. Example

    ```
    icmp type != { echo-reply, redirect }
    ```

  - Other usual matches are: tcp length, tcp checksum tcp flags . . . , udp length, udp checksum . . . icmp code, icmp checksum, icmp id . . .

nftables: rules matches

- most usual matches (**ct**)
  - **ct state stat** *stat* can be one in {`new, established, related, untracked, invalid`}, a set o a negation (!=). Example

    ```
    ct state != related
    ct state {new, established}
    ```

  - **ct direction dir**.*dir* can be {`original, reply`}
  - **ct status stat** *stat* can be one in {`expected,seen-reply,assured,confirmed,snat,dnat,dying`}, a set o a negation (!=). Example

    ```
    ct status expected
    ct status {snat,dnat}
    ```

nftables: rules statements

- **accept** Accept the packet and stop the remaining rules evaluation.
- **drop** Drop the packet and stop the remain rules evaluation.
- **queue** Queue the packet to userspace and stop the remain rules evaluation.
- **continue** Continue the ruleset evaluation with the next rule.
- **return** Return from the current chain and continue at the next rule of the last chain. In a base chain it is equivalent to accept
- **jump chain** Continue at the first rule of *chain*. It will continue at the next rule after a return statement is issued
- **goto chain** Similar to jump, but after the new chain the evaluation will continue at the next chain instead of the one containing the goto statement

nftables: rules statements

- **log [level lev]**. *lev* is one of the following: `emerg`, `alert`, `crit`, `err`, `warn`, `notice`, `info`, `debug`. Examples

```
log
log level crit
```

- **reject [with imcp type typ]**. *typ* is one of the following: `host-unreachable`, `net-unreachable`, `prot-unreachable`, `port-unreachable`, `net-prohibited`, `host-prohibited`, `admin-prohibited`. Examples

```
reject
reject with imcp type net-unreachable
```

nftables: rules statements

- **limit rate [over] value unit [burst value unit]**. Example

```
limit rate over 40/day
limit rate over 400/week
limit rate over 1023/second burst 10 packets
```

nftables: rules statements

- **dnat to destination_address [:port]**
- **snat to source_address [:port]**
- **masquerade** [to :port]
- **redirect** [to :port]
- The masquerade statement is a special form of snat which always uses the outgoing interface's IP address to translate to
- The redirect statement is a special form of dnat which always translates the destination address to the local host's one

nftables: scripting

- nft can be used in scripts. Here we have an example of script that creates a ipv4 firewall

```
#!/usr/sbin/nft -f
 flush ruleset

table firewall {
  chain incoming {
    type filter hook input priority 0; policy drop;
    # established/related connections
    ct state established,related accept
    # loopback interface
    iifname lo accept
    # icmp
    icmp type echo-request accept
   # open tcp ports: sshd (22), httpd (80)
    tcp dport {ssh, http} accept
  }
}
```

nftables: scripting

- The preivious script is squivalent to this script (or to executing those commands from the command line)

```
#!/bin/sh
nft flush ruleset
nft add table firewall
nft add chain firewall incoming '{type filter hook input priority filter;\
                        policy drop;}'
nft add rule firewall incoming ct state {established, related} accept
nft add  rule firewall incoming iifname "lo" accept
nft add  rule firewall incoming icmp type echo-request accept
nft add  rule firewall incoming tcp dport {22, 80} accept
```

nftables: scripting

- Should we want to create a script with the rules we have created at some point

```
# nft list ruleset > nules-nft
```

- we can now re-estblish that very same configuration with

```
# nft flush ruleset
# nft -f ./rules-nft
```

- or we can create a standalone script by adding these two lines at the begining of file rules-nft (and giving it execution permission)

```
#!/usr/sbin/nft -f

nft flush ruleset
```

Example: Securing the sshd server

sshd configuration

- ssh is the de facto standard for remote acces to machines, having made telnet, rlogin, rsh . . . obsolete
- its name stands for *secure shell* and the communication between the server and the client is crypted
- it is fairly secure, although protocol 1 had a serious vulnerability some years ago
- as is the tool of choice for accessing unix/linux servers we want to have it as secured as possible
- here are some tips. unless otherwise specified, they refer to options in the server's configuration file, `/etc/ssh/sshd_connfig`

securing sshd

- disable root login. We don't want the root to login directly onto the machine, so `ssh` is no exception

  ```
  PermitRootLogin no
  ```

- use only protocol 2, as protocol 1 has some known security holes

  ```
  Protocol 2
  ```

- force user into passwords policy and disable null passwords in ssh

  ```
  PermitEmptyPasswords no
  ```

- limit connections only to the machines you need to be able to connect from, using tcpwrappers (files `hosts.allow` and `hosts.deny`) and/or a firewall

securing sshd

- if possible, allow only some users to login (the ones who **actually** need to use the service)

```
AllowUsers user1 user2 user5
```

- put a maximun waiting time untill a connection happens

```
LoginGraceTime 60
```

- limiting the maximum number of concurrent connections to make brute force attacks more difficult

```
MaxStartups 2
```

- log off user after being idle sometime (in this example 6 minutes: 360 secs)

```
ClientAliveInterval 300
ClientAliveCountMax 0
```

securing sshd

- if possible use a non stardard port a listen only at one specific address (in case your machine has various addresses)

```
Port 2222
ListenAddress 192.168.0.10
```

- forwarding options

```
X11Forwarding no
AllowTcpForwarding no
```

- use some utility like `fail2ban` that blocks burte force authetication attempts
- if possible use a non password based authentication
- if possible use a 2 step authentication

Example: Securing the sshd server
→public/private key authetication

public/private key authetication

- we can use ssh to login remotely on a machine without having to send the password (which, although being crypted, could be vulnerable to *man in the middle* attacks)
- what we do is generate a pair of public/private keys. We keep our private key on our client machine, and place a copy of the public key on any of the server machines we want to connect to.
- we can now connect directly to those servers.

public/private key authetication

- `ssh-keygen -t rsa` generates the pair of keys. A passphase can be added to protect the key, otherwise any with access to our client machine will have direct access to our server machines
- the private key is `.ssh/id_rsa` and the public key is `.ssh/id_rsa.pub`
- the public key should be added to the file `.shh/authorized_keys` in the host
- `.ssh` directory should have permissions 0700 and any file in it should have 0600

Example: Securing the sshd server
→two step authenticator

google authenticator

- we install *google authenticator* app in our phone
- we install the pam module `pam_google_authenticator` (with `apt-get install libpam-google-authenticator`)
- each user must generate a key with `google-authenticator` to be read in the mobile app
- we add the following line to `/etc/pam.d/sshd` (nullok, to still allow users who had not generated the key to login)

```
auth required pam_google_authenticator.so nullok
```

- and have the following option in the *sshd* configuration file `ChallengeResponseAuthentication yes`