

Packet filtering with Linux

NETFILTER AND IPTABLES

Corso di Configurazione e Gestione di Reti Locali

Marco Bonola
Lorenzo Bracciale

A.A. 2011/2012

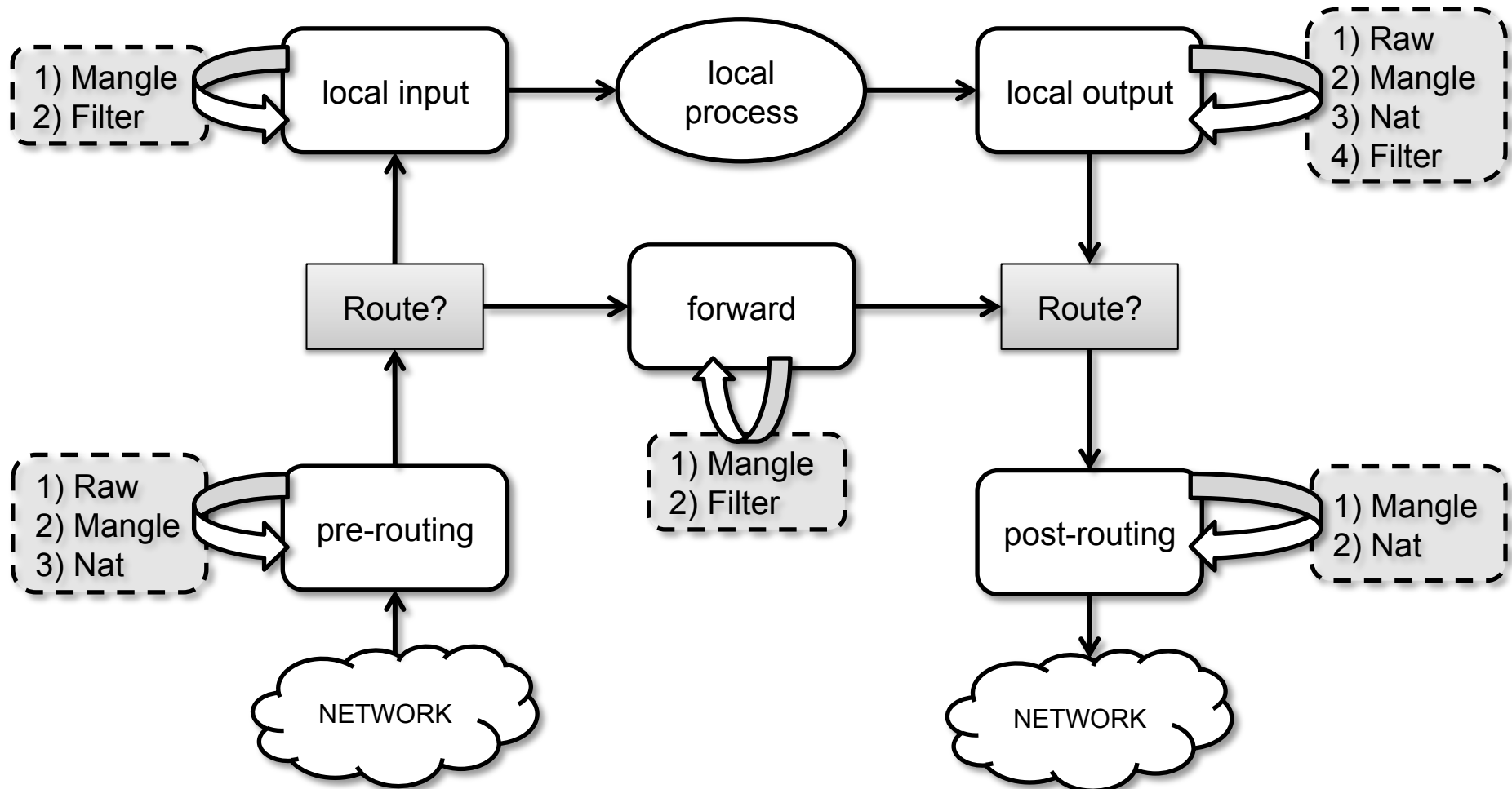
NETFILTER

- **NETFILTER** is a framework that provides hook handling within the Linux kernel for intercepting and manipulating network packets
- A **hook** is an “entry point” within the Linux Kernel IP (v4|v6) networking subsystem that allows packet mangling operations
 - Packets traversing (incoming/outgoing/forwarded) the IP stack are **intercepted** by these hooks, **verified** against a given set of matching rules and **processed** as described by an **action** configured by the user
- 5 built-in hooks:
 - PRE_ROUTING, LOCAL_INPUT, FORWARD, LOCAL_OUT, POST_ROUTING

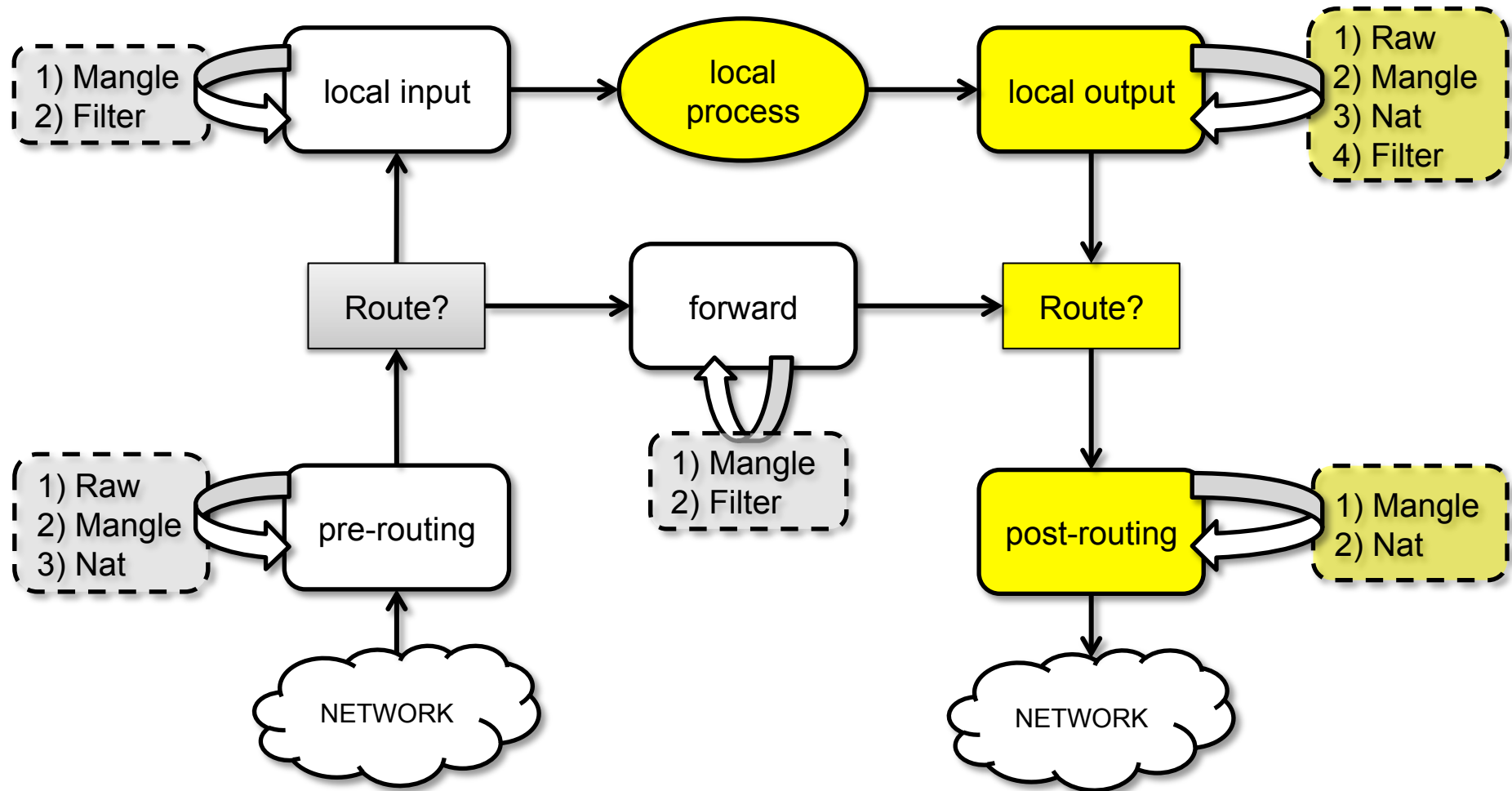
NETFILTER basics

- All packet intercepted by the hooks pass through a sequence of built-in **tables** (queues) for processing. Each of these queues is dedicated to a particular type of packet activity and is controlled by an associated packet transformation/filtering chain
- 4 built-in tables
 - **Filter**: packet filtering (accept, drop)
 - **Nat**: network address translation (snat, dnat, masquerade)
 - **Mangle**: modify the packet header (tos, ttl)
 - **Raw**: used mainly for configuring exemptions from connection tracking in combination with the NOTRACK target

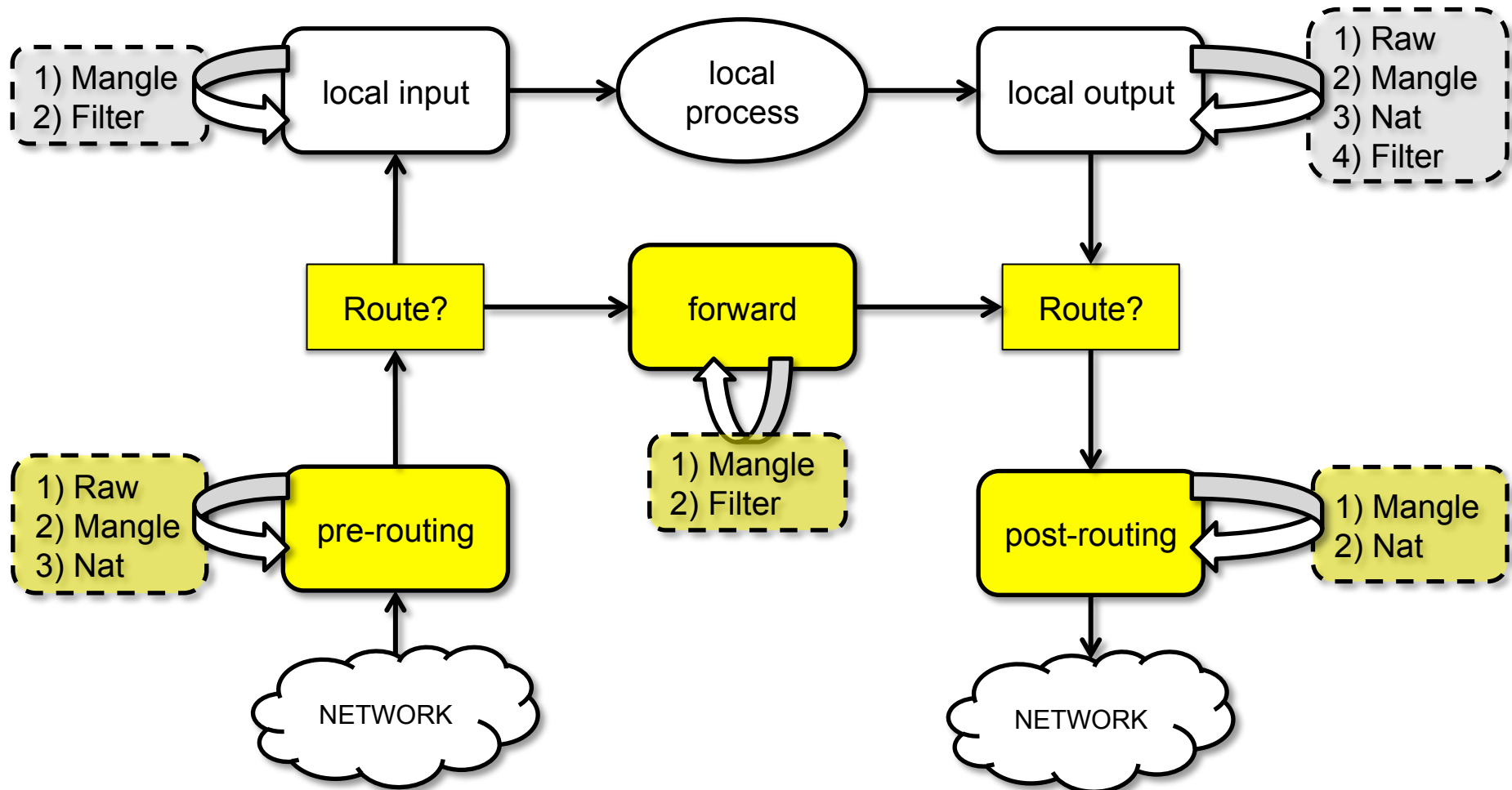
NETFILTER – The big picture



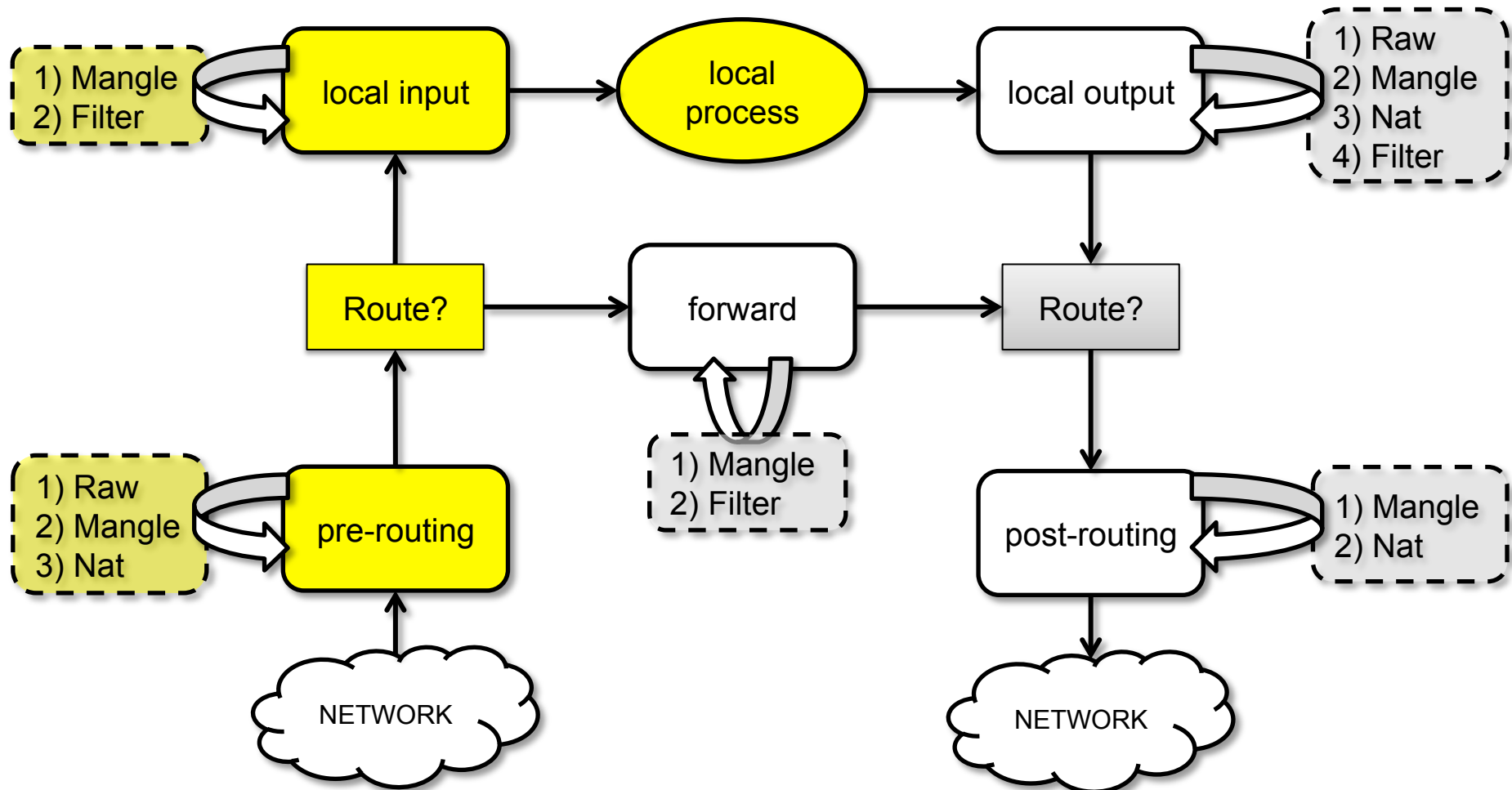
Locally generated packets



Forwarded packets



Locally addressed packets



NETFILTER basics

- The **matching rules** and the **actions (targets)** are implemented by different kernel modules and provides a powerful packet filtering system
- Common matches
 - Protocol, source/destination address or network, input/output interface, source/destination TCP/UDP port
- Common targets
 - ACCEPT, DROP, MASQUERADE, DNAT, SNAT, LOG
- NETFILTER is extensible
 - You can register your custom HOOK
 - You can write your own “matching module” and “action module”
 - http://jengelh.medozas.de/documents/Netfilter_Modules.pdf

Connection tracking system

- Within NETFILTER packets can be related to tracked connections in four different so called states
 - **NEW, ESTABLISHED, RELATED, INVALID**
- With the “state” match we can easily control who or what is allowed to initiate new sessions. More later on...
- To load the conntrack module
 - `modprobe ip_conntrack`
- `/proc/net/ip_conntrack` gives a list of all the current entries in your conntrack database

Connection tracking system

- `conntrack` util provides a full featured userspace interface to the netfilter connection tracking system that is intended to replace the old `/proc/net/ip_conntrack` interface
 - Commands: `dump`, `create`, `get`, `delete`, `update`, `event`, `flush`, `stats...`
 - `man conntrack`
 - `apt-get install conntrack`
- Two internal tables
 - **conntrack**: it contains a list of all currently tracked connections through the system
 - **expect**: it is the table of expectations. Connection tracking expectations are the mechanism used to "expect" RELATED connections to existing ones. Expectations are generally used by "connection tracking helpers" (sometimes called application level gateways [ALGs]) for more complex protocols such as FTP, SIP, H.323

NETFILTER conntrack

```
marlon@marlon-vmxnb:~$ sudo cat /proc/net/ip_conntrack
[sudo] password for marlon:

udp      17 28 src=172.16.166.156 dst=172.16.166.2 sport=43716 dport=53 src=172.16.166.2
dst=172.16.166.156 sport=53 dport=43716 mark=0 use=2

tcp      6 431951 ESTABLISHED src=172.16.166.156 dst=172.16.166.2 sport=48680 dport=9999
src=172.16.166.2 dst=172.16.166.156 sport=9999 dport=48680 [ASSURED] mark=0 use=2

udp      17 28 src=172.16.166.156 dst=172.16.166.2 sport=44936 dport=53 src=172.16.166.2
dst=172.16.166.156 sport=53 dport=44936 mark=0 use=2

udp      17 19 src=172.16.166.156 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED]
src=224.0.0.251 dst=172.16.166.156 sport=5353 dport=5353 mark=0 use=2

tcp      6 431487 ESTABLISHED src=172.16.166.156 dst=172.16.166.1 sport=43733 dport=139
src=172.16.166.1 dst=172.16.166.156 sport=139 dport=43733 [ASSURED] mark=0 use=2

udp      17 28 src=172.16.166.156 dst=172.16.166.2 sport=43581 dport=53 src=172.16.166.2
dst=172.16.166.156 sport=53 dport=43581 mark=0 use=2
```

conntrack events

```
root@marlon-vmxnb:/home/marlon# conntrack --event
[NEW] udp      17 30 src=172.16.166.156 dst=172.16.166.156 sport=47282 dport=4444 [UNREPLIED]
src=172.16.166.156 dst=172.16.166.156 sport=4444 dport=47282

[DESTROY] udp   17 src=172.16.166.2 dst=172.16.166.156 sport=5353 dport=5353 [UNREPLIED]
src=172.16.166.156 dst=172.16.166.2 sport=5353 dport=5353

[DESTROY] udp   17 src=172.16.166.156 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED]
src=224.0.0.251 dst=172.16.166.156 sport=5353 dport=5353

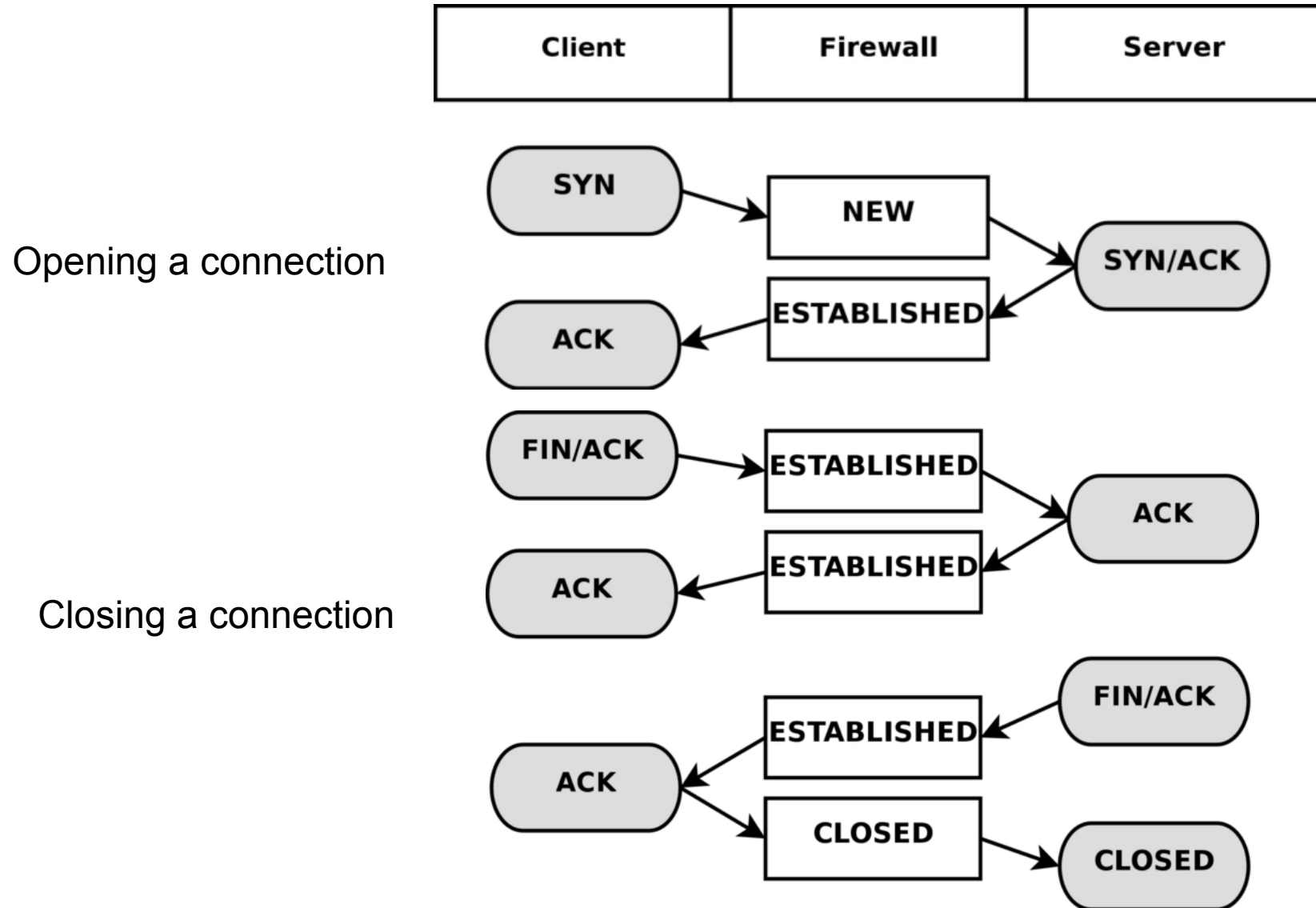
[DESTROY] udp   17 src=172.16.166.1 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED]
src=224.0.0.251 dst=172.16.166.1 sport=5353 dport=5353

[NEW] tcp      6 120 SYN_SENT src=172.16.166.156 dst=160.80.103.147 sport=45696 dport=80
[UNREPLIED] src=160.80.103.147 dst=172.16.166.156 sport=80 dport=45696

[UPDATE] tcp   6 60 SYN_RECV src=172.16.166.156 dst=160.80.103.147 sport=45696 dport=80
src=160.80.103.147 dst=172.16.166.156 sport=80 dport=45696

[UPDATE] tcp   6 432000 ESTABLISHED src=172.16.166.156 dst=160.80.103.147 sport=45696
dport=80 src=160.80.103.147 dst=172.16.166.156 sport=80 dport=45696 [ASSURED]
```

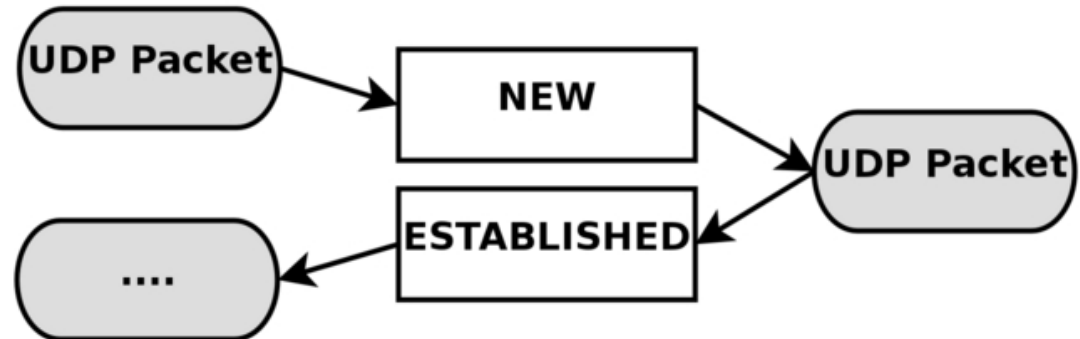
State machines - TCP



State machines - UDP

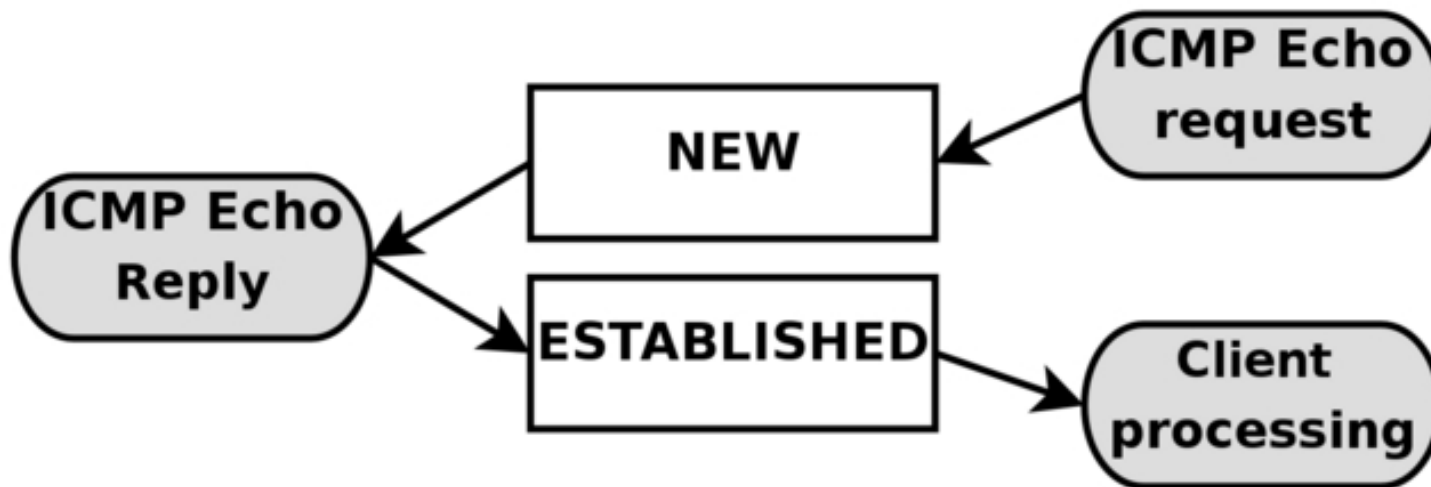


Opening a connection

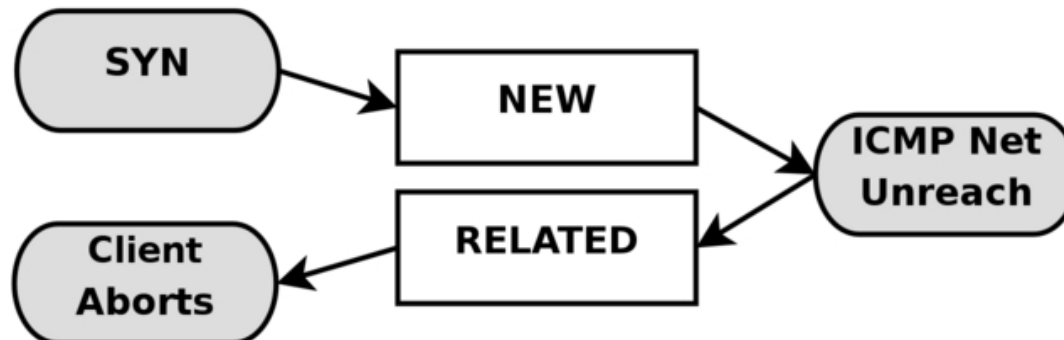
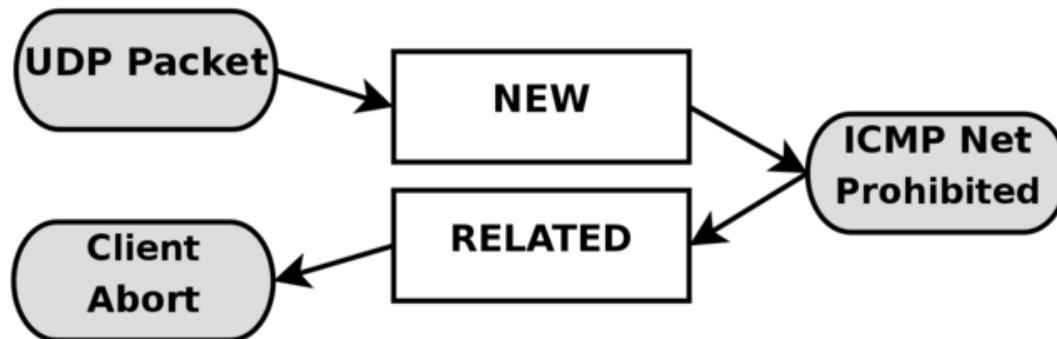


Closing a connection?

State machines - ICMP



State machines – ICMP related



ALG related state

- protocols like FTP, IRC, and others carry information within the actual data payload of the packets, and hence requires special connection tracking helpers to enable it to function correctly
- For example, FTP first opens up a single connection that is called the FTP control session and negotiate the opening of the data session over a different socket
- When a connection is done actively, the FTP client sends the server a port and IP address to connect to. After this, the FTP client opens up the port and the server connects to that specified port from a random unprivileged port (>1024) and sends the data over it
- A special NETFILTER conntrack helper can read the FTP control payload and read the “data port”
- The new data socket will be considered as RELATED
- Supported helpers
 - FTP, IRC, TFTP, SIP, etc..
 - <https://home.regit.org/netfilter-en/secure-use-of-helpers/>

IPTABLES

- **iptables** is the frontend of NETFILTER
- In other words, iptables is the userspace application used to configure the NETFILTER tables
- It is mainly used to add/remove rules to a chain (mapping of NETFILETR hooks) within a table

- General structure for adding remove a rule

```
iptables <command> <chain> <table> <match> <target>
```

```
iptables -A POSTROUTING -t nat -o eth0 -j MASQUERADE
```

- More later on...

IPTABLES TUTORIAL

Iptables

- **iptables** is used to set up, maintain, and inspect the tables of IPv4 packet filter rules in the Linux kernel
- Several different **tables** may be defined. Each table contains a number of built-in **chains** and may also contain user-defined chains
- Each chain is a list of rules which can **match** a set of packets
- Each rule specifies what to do with a packet that matches. This is called a **target**, which may be a jump to a user-defined chain in the same table

Iptables COMMANDS

- Append, delete, insert, replace rules
 - iptables [-t table] {-A|-D} chain rule-specification
 - iptables [-t table] -D chain rulenum
 - iptables [-t table] -I chain [rulenum] rule-specification
 - iptables [-t table] -R chain rulenum rule-specification
- List, flush rules
 - iptables [-t table] -S [chain [rulenum]]
 - iptables [-t table] -{F|L} [chain [rulenum]] [options...]
- Create, delete, rename chains and set 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
- Where:
 - rule-specification = [matches...] [target]
 - match = -m matchname [per-match-options]
 - target = -j targetname [per-target-options]

Iptables TARGETS

- A firewall rule specifies criteria for a packet and a target. If the packet does not match, the next rule in the chain is the examined;
- if the packet does match, then the next rule is specified by the value of the target (option -j), which 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 on the floor
 - **QUEUE** means to pass the packet to userspace
 - **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. More later on...

TABLES and CHAINS

- **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 and CHAINS

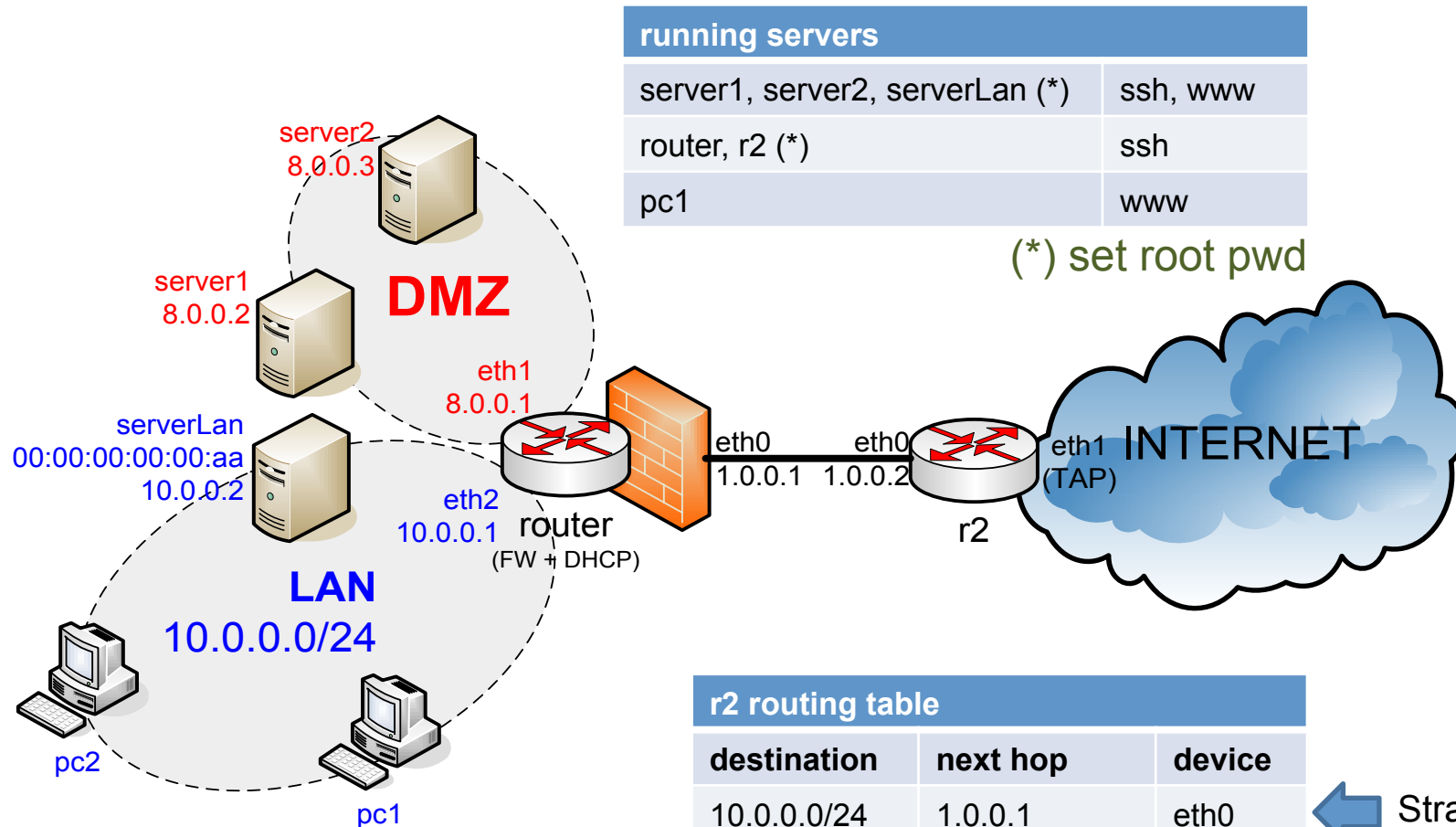
Queue Type	Queue Function	Packet Transformation Chain in Queue	Chain Function
Filter	Packet filtering	FORWARD	Filters packets to servers accessible by another NIC on the firewall.
		INPUT	Filters packets destined to the firewall.
		OUTPUT	Filters packets originating from the firewall
Nat	Network Address 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 match specification

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
- **[!] -f, --fragment**: This means that the rule only refers to second and further fragments of fragmented packets. Since there is no way to tell the source or destination ports of such a packet (or ICMP type), such a packet will not match any rules which specify them. When the "!" argument precedes the "-f" flag, the rule will only match head fragments, or unfragmented packets

Reference LAB – Lab5-nf



running servers

server1, server2, serverLan (*)	ssh, www
router, r2 (*)	ssh
pc1	www

(*) set root pwd

r2 routing table

destination	next hop	device
10.0.0.0/24	1.0.0.1	eth0
8.0.0.0/24	1.0.0.1	eth0
1.0.0.0/24	*	eth0
default	host-machine	eth1

← Strange...
Just to make
things work,
for now...

Our first simple commands

- Show rules in the filter table with numeric output and rule numbers

```
router# iptables -L -n --line-numbers
```

- Flush rules in the filter table

```
router# iptables -F
```

- Set the policy for the FORWARD chain in the filter table

```
router# iptables -P FORWARD DROP
```

- Allow all packets coming from LAN toward anywhere

```
router# iptables -A FORWARD -i eth2 -j ACCEPT
```

- Allow all packets from the Internet to DMZ

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

- Allow packets from server1 to LAN

```
router# iptables -A FORWARD -o eth2 -s 8.0.0.2 -i eth1 -j ACCEPT
```

Rule order is important...

- DROP all incoming packets from LAN except from serverLAN

```
router# iptables -A INPUT -s 10.0.0.0/24 -i eth2 -j DROP
router# iptables -A INPUT -s 10.0.0.2 -i eth2 -j ACCEPT
```

It won't work! Why?

Rule order is important...

- DROP all incoming packets from LAN except from serverLAN

```
router# iptables -A INPUT -s 10.0.0.0/24 -i eth2 -j DROP
router# iptables -A INPUT -s 10.0.0.2 -i eth2 -j ACCEPT
```

It won't work! Why?

Because the second rule is appended and then a packet from 10.0.0.2 will be dropped anyway as it will match the first rule

- Solutions:
 - Change the order ☺
 - Insert the second rule instead of appending it
 - set DROP as default policy for the INPUT chain and append the 1 ACCEPT rule

Save and restore

- To save a currently running iptables configuration
 - `iptables-save > firewall.conf`
- To restore a saved iptables configuration
 - `iptables-restore < firewall.conf`
- To automatically restore a configuration at startup use `/etc/rc.local`

- Debian based distros have a tool named `iptables-persistent` that provides a iptables service script
 - `apt-get install iptables-persistent`
 - `/etc/init.d/iptables-persistent start | stop | save | reload`
 - Configuration in `/etc/iptables/rules.v{4,6}`

Match extensions

- iptables can use extended packet matching modules
 - implicitly loaded when `-p` or `--protocol` is specified
 - or with `-m` or `--match` options, followed by the matching module name
- After an extended match is specified, various extra command line options become available, depending on the specific module
 - `iptables -A INPUT -p tcp --sport 9000 -j DROP`
 - `iptables -A INPUT -m addrtype --dst-type MULTICAST -j DROP`
- You can specify multiple extended match modules in one line, and you can use the `-h` or `--help` options after the module has been specified to receive help specific to that module
- Man iptables for all match extensions

TCP and UDP match extensions

- `-p tcp`: matches IP packets with protocol=TCP
 - `--sport`: matches the TCP source port
 - `--dport`: matches the TCP destination port
 - `--tcp-flags`: matches the TCP header flags...
 - `--syn`: matches packets with the SYN flag set

Example:

```
iptables -A INPUT -p tcp --dport 80 -j DROP
```

- `-p udp`: matches IP packets with protocol=UDP
 - `--sport`: matches the UDP source port
 - `--dport`: matches the UDP destination port

Example:

```
iptables -A OUTPUT -p udp --dport 53 -j DROP
```

Example: forward SSH traffic

Allow forwarding of SSH traffic from clients inside the LAN and servers on the internet

```
router# iptables -A FORWARD -i eth2 -p tcp --dport 22 -j ACCEPT
```

```
router# iptables -A FORWARD -i eth0 -p tcp --sport 22 -j ACCEPT
```

We would like to accept traffic from the internet with source port 22 only if related to a previously established connection from LAN...

State match extensions

This module, when combined with connection tracking, allows access to the connection tracking state for this packet

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

Where *state* is a comma separated list of the connection states to match.

- **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

-m ctstate provides additional features. Man iptables for more...

Examples: state match

- Forward traffic between LAN and INTERNET if initiated from LAN

```
router# iptables -P FORWARD DROP
router# iptables -A FORWARD -i eth2 -m state --state
NEW, ESTABLISHED -j ACCEPT
router# iptables -A FORWARD -i eth0 -m state --state
ESTABLISHED -j ACCEPT
```

- Accept incoming/outgoing traffic only if related to locally initiated traffic. Accepts only incoming connections **ONLY** for SSH (std port)

```
router# iptables -P INPUT DROP
router# iptables -P OUTPUT DROP
router# iptables -A OUTPUT -m state --state NEW,
ESTABLISHED -j ACCEPT
router# iptables -A INPUT -m state --state
ESTABLISHED -j ACCEPT
router# iptables -A INPUT -p tcp --dport 22 -m state
--state NEW -j ACCEPT
```

Example: save and restore

- With respect to the previous slide
 - save the iptables configuration somewhere
 - configure router VM to restore this configuration at startup
 - reboot the VM
 - verify that the iptables configuration still works

Multiport match

- Match multiple destination ports (e.g. tcp)
 - `-p tcp -m multiport --dports port1,port2,...,portn`
- Match multiple source ports (e.g. udp)
 - `-p udp -m multiport --sports port1,port2,...,portn`
- Match multiple ports (both src and dst) (tcp)
 - `-p tcp -m multiport --ports port1,portb:portc`

Target extensions

- iptables can use extended target modules
- Target modules are automatically loaded when -j option is specified
- Common targets
 - DNAT, SNAT, MASQUERADE, REDIRECT
 - Network address translation. later on....
 - LOG
 - Log the matching packets. See `/var/log/syslog/` or run `dmesg`
 - `--log-prefix`: specify a prefix string
 - MARK
 - Internally set a mark to the matching packet
 - `-j MARK --set-mark <u32>` #set mark target
 - `-m mark --mark <u32>` #mark match
 - REJECT
 - Drop a matching packet and send a specific error message

Example: mark and log

As (a stupid) example, let's mark all TCP syn and all "new" UDP packets locally addressed and log them (first flush everything and set policies to ACCEPT)

```
router# iptables -F && iptables -P INPUT ACCEPT &&  
iptables -P OUTPUT ACCEPT && iptables -P FORWARD  
ACCEPT
```

```
router# iptables -A INPUT -m state --state NEW -j  
MARK --set-mark 1234
```

```
router# iptables -A INPUT -m mark --mark 1234 -j LOG  
--log-prefix "test-log "
```


User defined chains

It is possible to define custom chains. It is useful to:

- keep a big configuration in order
- reduce the number of rules
- change default policies (NO POLICY for custom chains. Use the LAST rule)

To define a custom CHAIN:

```
iptables -N NEW_CHAIN
```

Add a rule to a custom chain:

```
iptables -A NEW_CHAIN -j LOG
```

To pass a packet to a custom chain (e.g. from the INPUT chain, filter table, source address 10.0.0.0/24):

```
iptables -A INPUT -s 10.0.0.0/24 -j NEW_CHAIN
```

To destroy a custom chain (be sure there are no rules pointing to this chain):

```
iptables -X NEW_CHAIN
```

Example of a MANGLE table target

Set TOS for ssh (standard port) to 1, udp traffic to 2, the remaining tcp traffic to 2 for packets sent through eth0 (both forwarded and locally generated)

```
# first flush everything and set all policies to ACCEPT #
router# iptables -t MANGLE -A POSTROUTING -o eth0 -p tcp -
j TOS --set-tos 3
router# iptables -t MANGLE -A POSTROUTING -o eth0 -p tcp
--dport 22 -j TOS --set-tos 1
router# iptables -t MANGLE -A POSTROUTING -o eth0 -p udp -
j TOS --set-tos 2
```

TEST it with tcpdump on r2

What is the difference if you want to do that only for packets forwarded from LAN ?

And only for packets locally generated?

Homework: firewall spec

- 1) Forward traffic between DMZ and the INTERNET
- 2) Forward traffic between LAN and DMZ only if initiated from LAN
- 3) Forward ssh, www and dns traffic between LAN and INTERNET only if initiated from LAN
- 4) Drop all traffic initiated from INTERNET to router except ssh and icmp (only ping)
- 5) Allow traffic from router to anywhere
- 6) DROP all traffic from LAN to router except ssh, dhcp, icmp, udp destination port 666 and TCP ports 10000, 20000, 30000
- 7) LOG all “accepted” packets

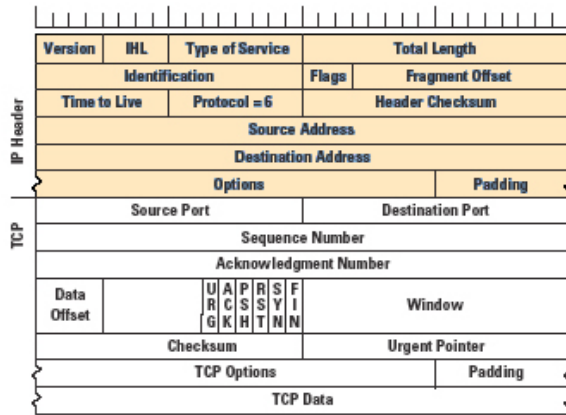
- 8) Save the script in a file so that it will work restored on any “fulshed” router

NETWORK ADDRESS TRANSLATION WITH NETFILTER

Network Address Translation

- **NAT** is the process of changing the the IP header
 - E.g.: a routed packet is intercepted, the IP source address is changed and the IP and L4 checksum in updated
 - RFC 2663 defines it as basic NAT or one to one NAT
- Since a static **one-to-one NAT** can't be exploited by an entire address space (e.g.: a 10.0.0.0/24 LAN behind NAT), the L4 ports can be changed to avoid ambiguity in the response packets
 - RFC 2663 also defines a “Network Address and Port Translation” (**NAPT**)
 - It is also referred to as **PAT, Masquerading, Many to One NAT** etc...
- **NAPT** technique allows to successfully forward packets addressed to the masqueraded network (e.g.: the LAN behind the NAT) only if related to a flow originated from it
 - For flows originated outside the masqueraded network, a different method is used
 - **Static NAT (or Port Forwarding)** uses static binding between $(addr_e:port_e) \leftrightarrow (addr_i:port_i)$
- **To much confusion!** From now on, we'll use “NAT” for the general IP/L4 translation

NAT basic mechanism - masquerading



ip_H : host private IP address
 ip_S : server public IP address
 ip_{NAT} : NAT public IP address
 tcp_{sport} : host application source port
 tcp_{dport} : server listening port
 tcp_{NAT} : random port picked up by NAT
 ip_{hc}, tcp_{cs} : protocol checksums

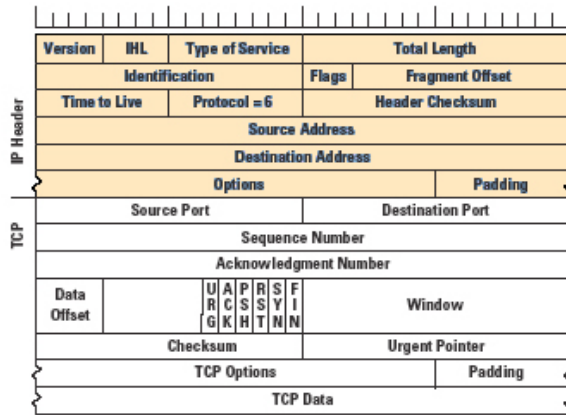
LAN host

NAT

server



NAT basic mechanism port forwarding



ip_S : server private IP address
 ip_C : client IP address
 IP_{NAT} : NAT public IP address
 tcp_{sport} : client application source port
 tcp_{dport} : server listening port
 tcp_{NAT} : port forwarded by NAT
 ip_{hc}, tcp_{cs} : protocol checksums

LAN host

NAT

client

Statically configured with
 $ip_H, tcp_{sport} : ip_{NAT} tcp_{NAT}$

$ip_C, ip_{NAT}, ip_{hc}, tcp_{sport}, tcp_{NAT}, tcp_{cs}$

$ip_C, ip_S, ip_{hc}^*, tcp_{sport}, tcp_{dport}, tcp_{cs}^*$

retrieve ip_S, tcp_{dport}
 from ip_{NAT}, tcp_{NAT}



More about NAT nomenclature

- NAT classification is really confusing among vendors:
 - CISCO
 - Static, dynamic
 - IBM
 - Static, dynamic, masquerading
 - LINUX (NETFILTER)
 - DNAT, SNAT, MASQUERADE, REDIRECT
 - JUNIPER
 - Full cone, symmetric
 - BSD
 - No explicitly distinct types
- From now on we'll use the **Linux** nomenclature

Address:Port binding strategy classification

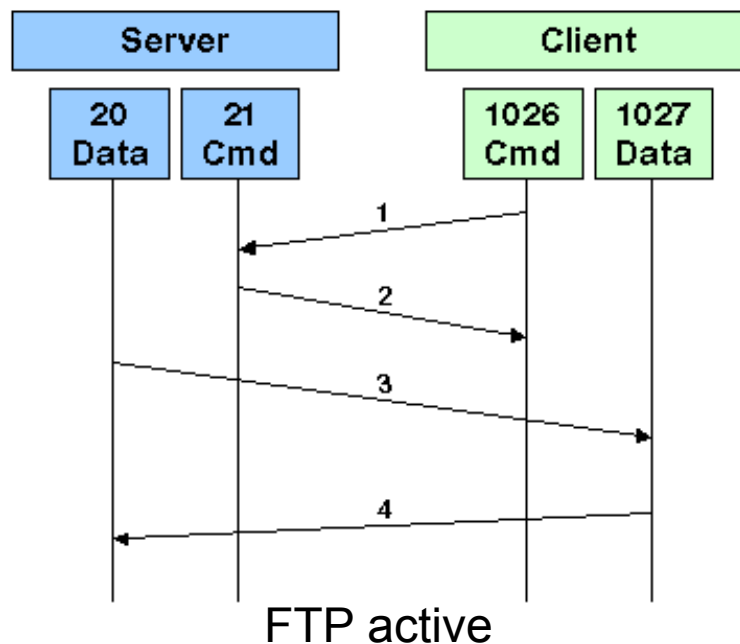
- RFC 3849 (Simple Traversal of UDP over NATs, obsoleted by RFC 5389) defines 4 different strategies for selecting the binding between $(addr_e:port_e) \leftrightarrow (addr_i:port_i)$
 - Full-cone NAT
 - Any external host can send packets to $iAddr:iPort$ by sending packets to $eAddr:ePort$
 - Restricted cone NAT
 - An external host ($hAddr:any$) can send packets to $iAddr:iPort$ by sending packets to $eAddr:ePort$ only if $iAddr:iPort$ has previously sent a packet to $hAddr:any$
 - Port-restricted cone NAT
 - An external host ($hAddr:hPort$) can send packets to $iAddr:iPort$ by sending packets to $eAddr:ePort$ only if $iAddr:iPort$ has previously sent a packet to $hAddr:hPort$
 - Symmetric NAT
 - Only an external host that receives a packet from an internal host can send a packet back
- As described in RFC 4787, this classification is inadequate to describe real NAT implementation (as they can be a mix of the above techniques. E.g: NETFILTER DNAT is fullcone, MASQUERADE is symmetric)
- Some NAT traversal protocols simply make the distinction Symmetric/Asymmetric NAT

Why NATs are bad (1)

- NATs are bad for servers in masqueraded LAN
- OBVIOUS! Without the NAT enabled router taking ad-hoc static port forwarding, how can a client reach such servers?
- Sometimes we can't control the NAT enabled router
 - E.g: FASTWEB home customers
- Even with total control over the NAT and static port forwarding, what if I have multiple (e.g. HTTP) servers?
 - Use different ports
 - How do I advert this?

Why NATs are bad (2)

- NATs are bad for clients too
- Example: FTP (textual protocol over TCP) ACTIVE MODE
- Other examples: RTP/RTCP, SIP, p2p protocols...



0. Connection from rand port and authentication
1. Client selects a port (rand+1) for receiving data traffic (PORT command)
2. Server acknowledges it
3. Server starts the connection to client:1027
4. Client acknowledges the SYN

Problems with a NAT in the middle:

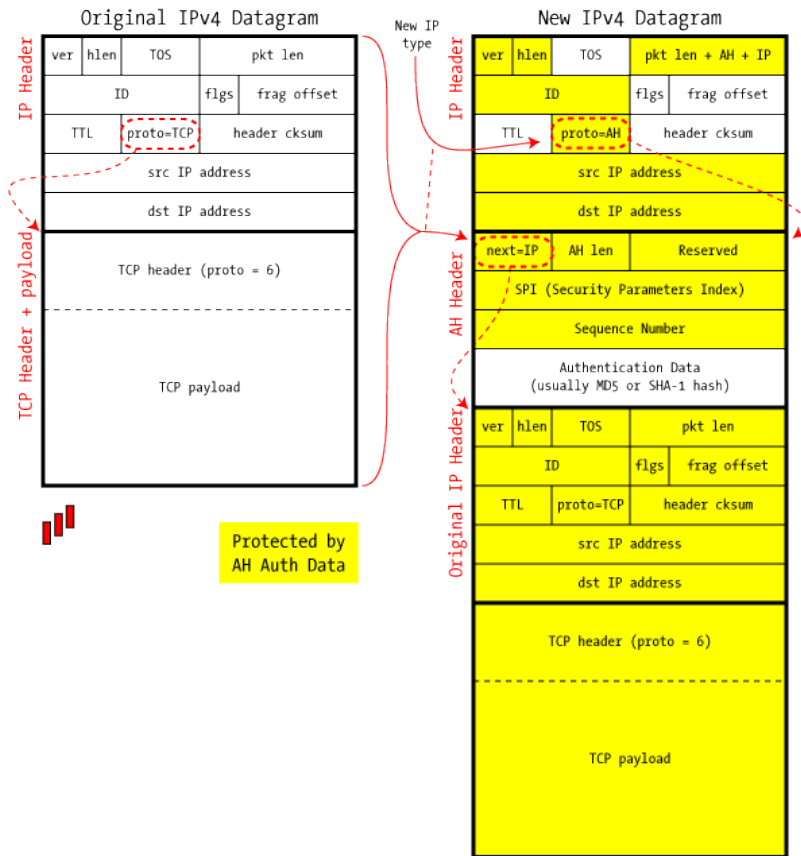
- The server contacts the client at the address it sees in the CMD connection (i.e: the public address of the NAT) and PORT in the FTP msg
- CMD and DATA ports are random
- How can the NAT correctly forward the DATA connection to the Client?
- Even if the client port were fixed, what would happen with multiple client behind the NAT?

Why NATs are bad (3)

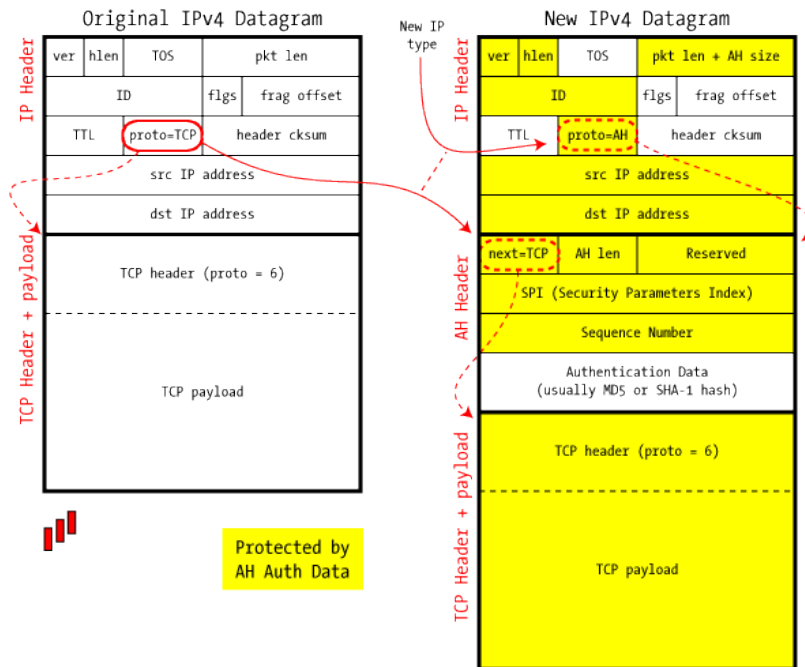
- NATs are bad for secure end to end encapsulation with IPSEC
- If AH (Authentication Header) is used, the NAT destroys the end to end cryptographic integrity computed over the whole IP packet
 - The NAT would change the source IP address
 - The NAT couldn't re-compute the message authentication code because he doesn't know the secret (as it should be...)
- Even with ESP only, what if multiple hosts selects the same SPIs?
 - To revert the binding, the NAT can use the couple (IP, SPI)
 - SPIs are picked up independently by the two parties
- NAT implementations are (not rarely) able to work only with IP +{TCP|UDP}
- NAT binding timeout
- Other motivations (described in RFC 3715)

Why NATs are bad (3)

IPSec in AH Tunnel Mode

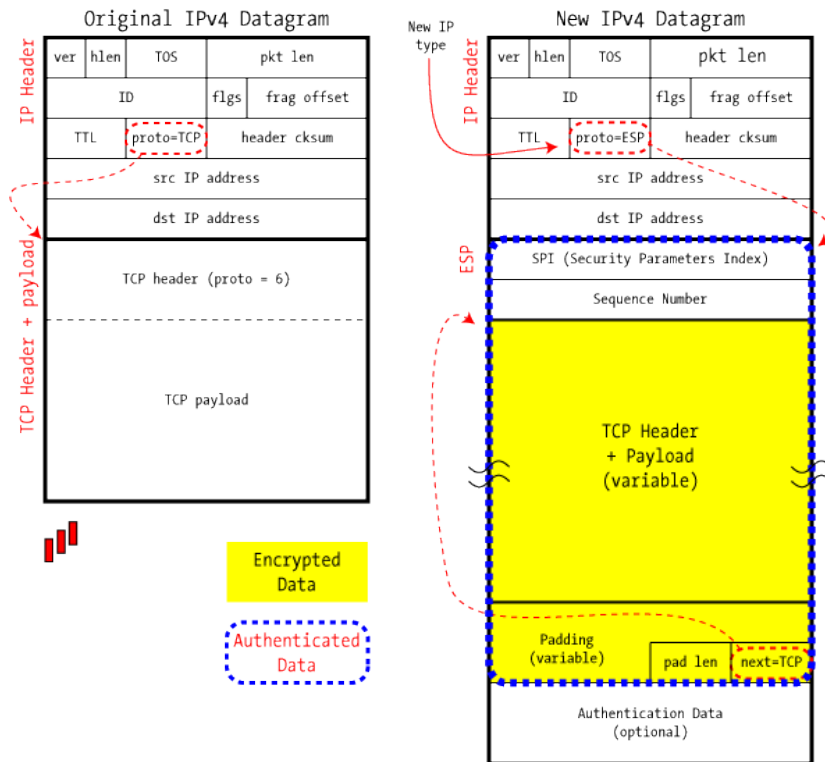


IPSec in AH Transport Mode

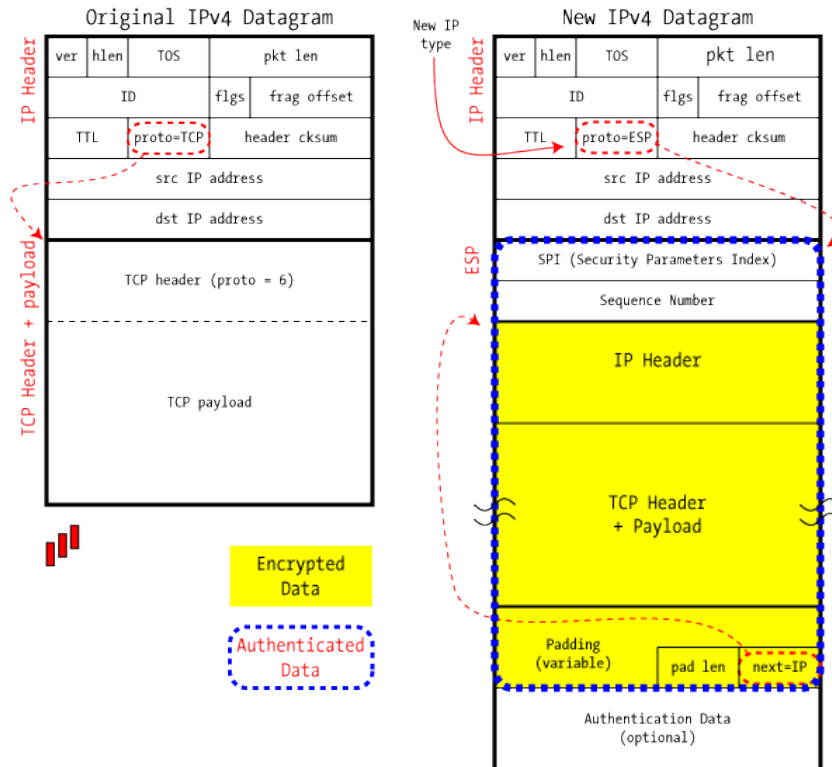


Why NATs are bad (3)

IPSec in ESP Transport Mode



IPSec in ESP Tunnel Mode



What is NAT traversal?

- NAT traversal is a mechanisms to solve the problems introduced by NATs
- It can be implemented as:
 - A protocol “patch” e.g: FTP passive
 - In FTP passive the client sends the first packet to allow the server to find out the DATA port
 - An ALG (application layer gateway) that inspects and changes the application protocols message
 - An integrated “helper” protocol used at set up time (Internet Connectivity Establishment (ICE), TURN, STUN)
 - Tunneling mechanism (e.g.: UDP encapsulation)

What are NATs good for?

1. IPv4 address exhaustion

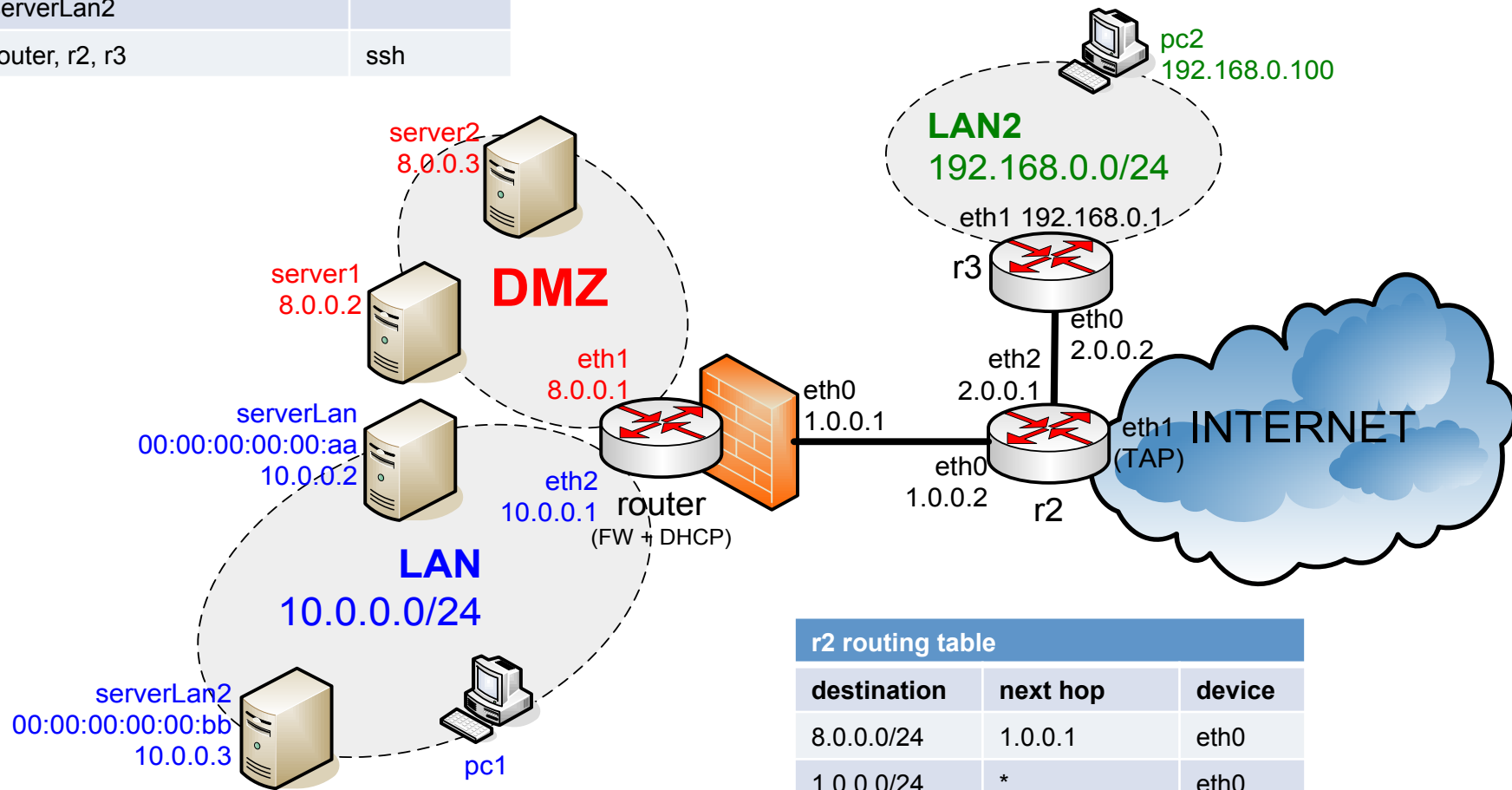
- NAT allows several hosts in a LAN to share the Internet connection through the same public IP

2. Implicit security

- A host in a LAN behind NAT can't be reached from outside...
- ...unless explicit port forwarding is configured

Lab5-nf-bis

running servers	
server1, server2, serverLan, serverLan2	ssh, www
router, r2, r3	ssh



r2 routing table		
destination	next hop	device
8.0.0.0/24	1.0.0.1	eth0
1.0.0.0/24	*	eth0
2.0.0.0/24	*	eth1
default	host-machine	TAP

SNAT

- This target is used to do Source Network Address Translation, which means that this target will rewrite the Source IP address in the IP header of the packet and the source L4 port (if needed)
- The SNAT target is only valid within the **nat table**, within the **POSTROUTING** chain
- Only the first packet in a connection is mangled by **SNAT**, and after that all future packets using the same connection will also be **SNAT**ted
- Syntax
 - `-j SNAT`
 - `--to-source ipaddr[-ipaddr][:port[-port]]`
 - `--random`
 - `--persistent`
 - Port range only valid with `[-p tcp|udp]` option
- Default behaviour: L4 ports are not modified, if they can be left unchanged (i.e.: if the specific port has not been already allocated for another binding)

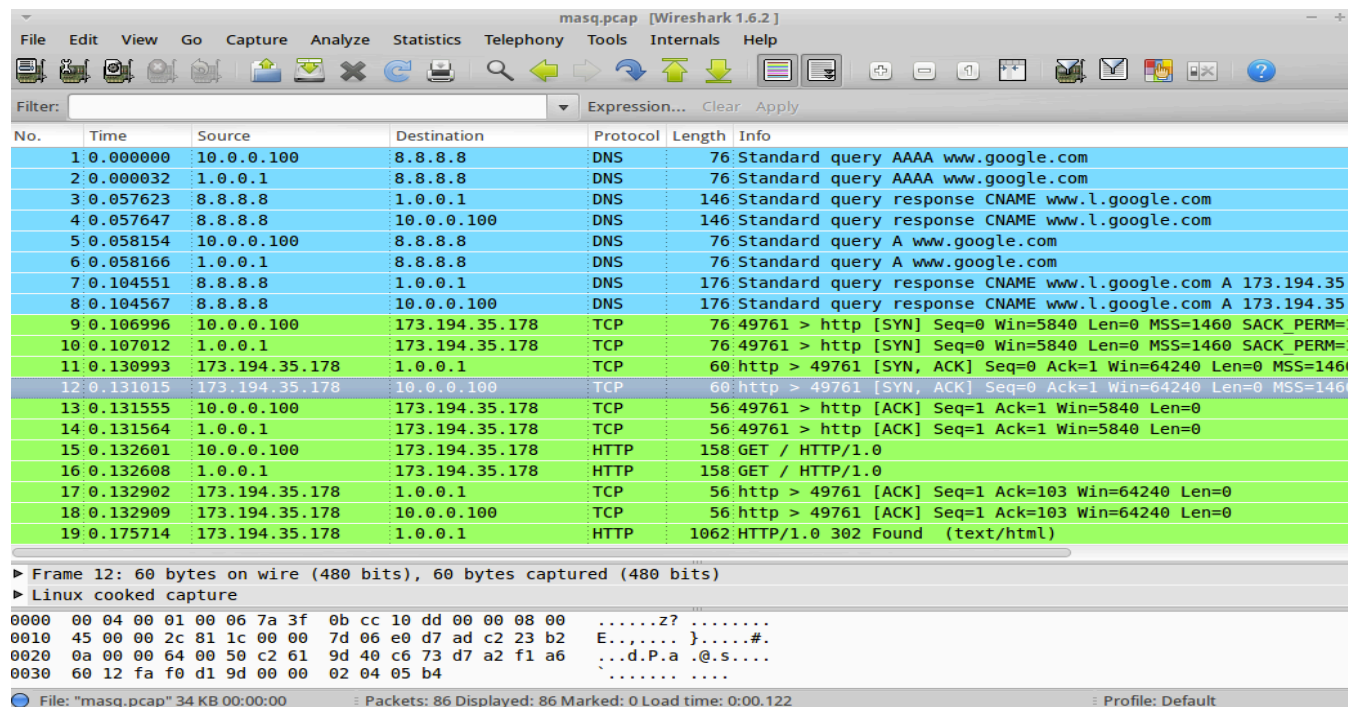
MASQUERADE

- The **MASQUERADE** target is used basically the same as the **SNAT** target, but it does not require any **--to-source** option
- **MASQUERADE** target was made to work with, for example, dial-up connections, or DHCP connections, which gets dynamic IP addresses when connecting to the network in question
- If you have a static IP connection, you should instead use the **SNAT** target
- Source address is dynamically grabbed from the output interface (it depends on the IP forwarding process)
- This target is only valid in the **nat table**, in the **POSTROUTING chain**
- Syntax
 - `-j MASQUERADE`
 - `--to-ports port[-port]`
 - `--random`
 - Port range only valid with `[-p tcp|udp]` option

MASQUERADE in Lab5-nf-bis

Configure router to MASQUERADE traffic going out from eth0
(r3 already run the following iptables command in r3.startup)

```
router# iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```



Since in the reference LAB eth0 is configured statically, it would have been equivalent (the manual says it's actually better for performances) to use SNAT from 1.0.0.1 (homework: verify it!)

```
router# iptables -t nat -A POSTROUTING -o eth0 -j SNAT --to-source 1.0.0.1
```

DNAT

- OK, host in LAN are now masqueraded and they can “access the INTERNET” sharing the public IP address of router
- What if I tried to reach the http server on ServerLan through the public IP on router?
 - I would get a TCP reset (it's not a rule...)
- Why?
 - Because router doesn't have a binding between 1.0.0.1:80 ↔ 10.0.0.2:80 and router thinks the TCP SYN received is for himself...
- What can I do?
 - Port forwarding (using NETFILTER DNAT)

DNAT

- This target is used to do Destination Network Address Translation, which means that it is used to rewrite the Destination IP address of a packet and the destination L4 port (if required)
- Note that the **DNAT** target is only available within the PREROUTING and OUTPUT chains in the nat table
- Syntax
 - `-j DNAT`
 - `--to-destination [ipaddr] [-ipaddr] [:port [-port]]`
 - `--random`
 - `--persistent`
 - Port range only valid with `[-p tcp|udp]` option

DNAT in Lab5-nf-bis

Let's make the http server on serverLan available through router public address and port 80

```
router# iptables -t nat -A PREROUTING -d 1.0.0.1 -p tcp --dport 80 -j  
DNAT --to-destination 10.0.0.2:80
```

Try from pc2

```
pc2# links 1.0.0.1
```

What about HTTP server on serverLan2?

We use another port (e.g: 8080)!

```
router# iptables -t nat -A PREROUTING -d 1.0.0.1 -p tcp --dport 8080 -  
j DNAT --to-destination 10.0.0.3:80
```

Try from pc2

```
pc2# links 1.0.0.1:8080
```

Note the double NAT from pc2 to serverLan[2]

DNAT for hosts in the same LAN

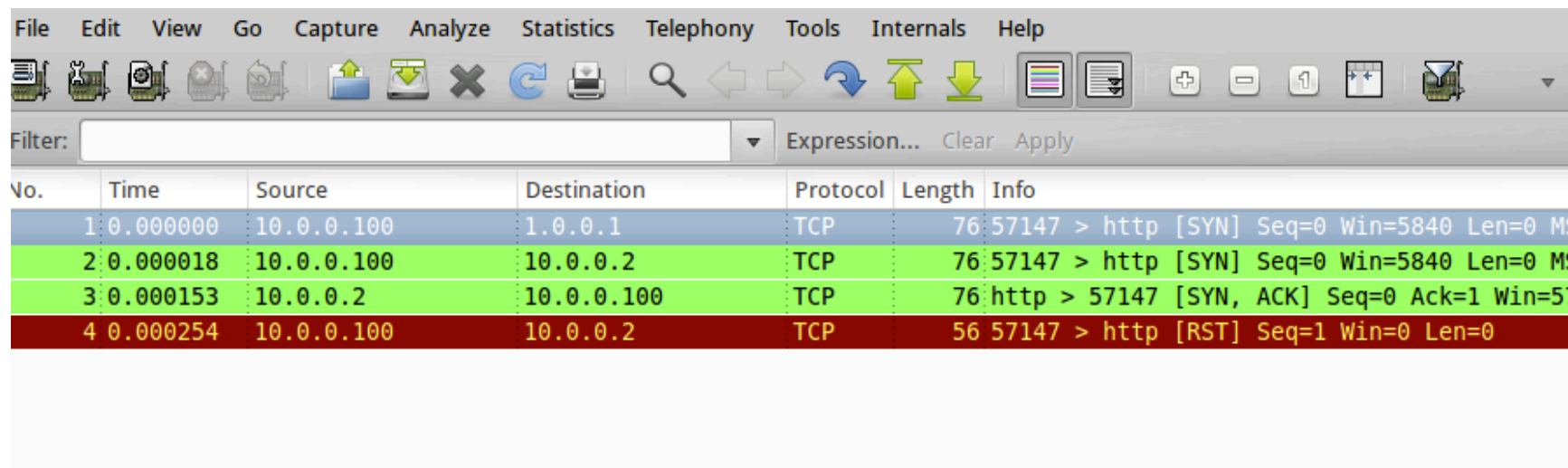
What if pc1 tries to connect to serverLan through 1.0.0.1:80?

- It won't work!

What happens?

- I have a TCP reset from 10.0.0.100 to 10.0.0.2

Huh? Really?



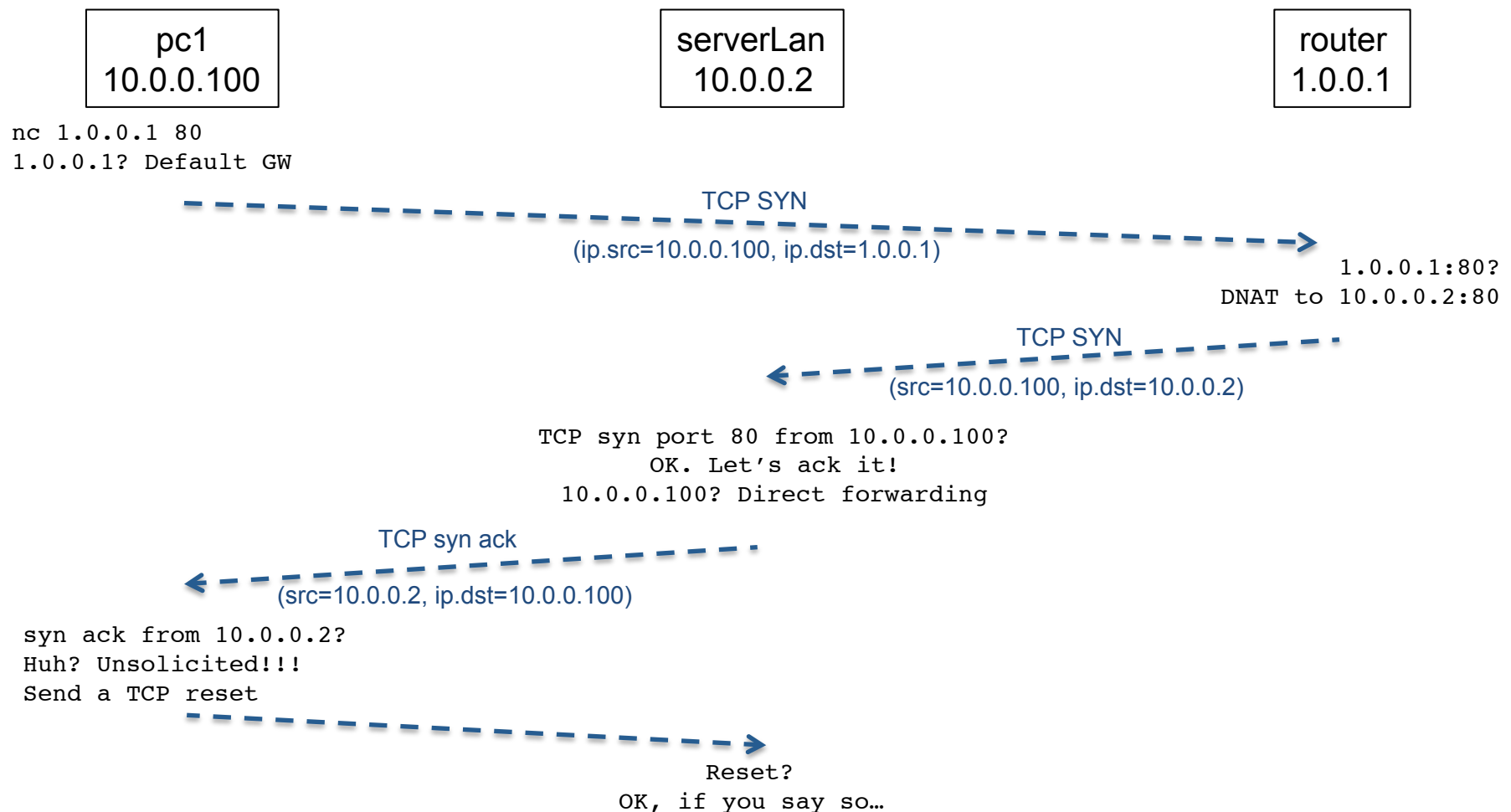
The image shows a Wireshark network traffic capture. The interface includes a menu bar (File, Edit, View, Go, Capture, Analyze, Statistics, Telephony, Tools, Internals, Help), a toolbar with various icons, and a filter field. The main display area shows a list of captured packets with the following details:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.0.100	1.0.0.1	TCP	76	57147 > http [SYN] Seq=0 Win=5840 Len=0 MS
2	0.000018	10.0.0.100	10.0.0.2	TCP	76	57147 > http [SYN] Seq=0 Win=5840 Len=0 MS
3	0.000153	10.0.0.2	10.0.0.100	TCP	76	http > 57147 [SYN, ACK] Seq=0 Ack=1 Win=5
4	0.000254	10.0.0.100	10.0.0.2	TCP	56	57147 > http [RST] Seq=1 Win=0 Len=0

DNAT for hosts in the same LAN

What happened?

(let's assume all mac:ip already in ARP cache)



DNAT for hosts in the same LAN

Solution

MASQUERADE packets to 10.0.0.2:80 from 10.0.0.0/24

```
router# iptables -t nat -A POSTROUTING -s 10.0.0.0/24 -d  
10.0.0.2 -p tcp --dport 80 -j MASQUERADE
```

Load balancing with DNAT

- GOAL: balance the connections to 1.0.0.1:80 equally between serverLan and serverLan2
- statistics match
 - --mode nth --every n: matches every n packets

```
router# iptables -t nat -A PREROUTING -d 1.0.0.1  
-p tcp --dport 80 -m statistic --mode nth --  
every 2 -j DNAT --to-destination 10.0.0.2:80
```

```
router# iptables -t nat -A PREROUTING -d 1.0.0.1  
-p tcp --dport 80 -m statistic --mode nth --  
every 1 -j DNAT --to-destination 10.0.0.3:80
```

REDIRECT

- This target is used to redirect packets and streams to the machine itself by changing the destination IP to the primary address of the incoming interface (locally-generated packets are mapped to the 127.0.0.1 address)
- Example: **REDIRECT** all packets destined for the HTTP ports to an HTTP proxy like squid
- **REDIRECT** target is only valid within the PREROUTING and OUTPUT chains of the nat table
- Syntax
 - -j REDIRECT
 - --to-ports port[-port]
 - --random
 - If --to-port is missing, the destination port is left unchanged
- Rule example (assuming there is a local proxy server listening on 8080)
 - `iptables -t nat -A PREROUTING -p tcp --dport 80 -j REDIRECT --to-ports 8080`
- Test: redirect http to server2 and server1 to a local server web
 - Run apache on router and add the following rule (on router)
 - `router# iptables -t nat -A PREROUTING -p tcp --dport 80 -j REDIRECT`
 - Run the following command on (for example) pc2
 - `pc2# links 8.0.0.2`

NETFILTER: SOME ADVANCED FEATURES

Limit matches

- This module matches at a limited rate using a token bucket filter
- A rule using this extension will match until this limit is reached (unless the '!' flag is used)
- Can be used in combination with other modules
- Syntax
 - -m limit
 - --limit *rate*[/**second**/*minute*/*hour*/*day*]
 - --limit-burst *number* This value indicates that the limit/minute will be enforced only after the total number of connection have reached the limit-burst level (*in other word is the token-bucket size*)
- Hashlimit provides extra features... (man iptables)
- **Example:** limit incoming http connections to 6 per minute after a maximum burst of 3 is reached (stupid numbers, but it's easy to check if it's working)

```
server1# iptables -P INPUT ACCEPT
```

```
server1# iptables -A INPUT -p tcp --dport 80 --syn -m  
limit --limit 6/minute --limit-burst 3 -j ACCEPT
```

```
server1# iptables -A INPUT -p tcp --syn -j DROP
```

Setting transfer quotas

- Set the maximum number of bytes allowed for a given match
- Ex: subscription bytes limit
- Explicitly delete and insert the rule to reset quotas (e.g.: use a cron job)
- Need to explicitly remove and re-insert the rule to reset quotas. (warning: no way to save the status)

Example: limit monthly quota to 2 GB

```
#configure iptables with
iptables -P INPUT DROP
iptables -A INPUT -m mark --mark 12 -j ACCEPT
```

```
#In file /root/quota.sh (make it executable)
iptables -D INPUT -m quota --quota 2000000000 -j MARK --set-mark 12
iptables -I INPUT -m quota --quota 2000000000 -j MARK --set-mark 12
```

```
#In file /etc/crontab add the following line
@monlty root /root/quota.sh
```

```
#start crontab
/etc/init.d/cron start
```

Example2: limit the quota 1 MB per minute (stupid numbers just to test it! Server1 has 2 files: small.jpg, medium and big.jpg)

```
#same as before but set quota to 1000000 and instead the wildcard @monthly use
the following line
```

```
* * * * * root/root/quota.sh
```

Time based rules

- We can match rules based on the time of day and the day of the week using the time module
- This could be used to limit staff web usage to lunch-times, to take each of a set of mirrored web servers out of action for automated backups or system maintenance, etc
- Example: allows web access during lunch hours

```
router# iptables -A FORWARD -p tcp -m multiport --dport  
http,https -o eth0 -i eth2 -m time --timestart 12:30 --  
timestop 13:30 --days Mon,Tue,Wed,Thu,Fri -j ACCEPT
```


String match

- This match identify a packet containing a string anywhere in it's payload
- Syntax
 - -m string
 - --string "*string*"
 - --from *offset*
 - --to *offset*
 - --algo *algorithm {bm|kmp}*

Example: LOG all HTTP GETs from LAN and DROP all ones containing "sex" in the URL
NOTE: turn on masquerading

```
router# iptables -P FORWARD ACCEPT && iptables -F
```

```
router# iptables -N HTTP_GET
```

```
router# iptables -A HTTP_GET -j LOG --log-prefix "HTTP GET "
```

```
router# iptables -A HTTP_GET -m string --string --algo bm "sex" -j  
DROP
```

```
router# iptables -A FORWARD -i eth2 -m string --algo bm --string "GET  
" --to 4 -j HTTP_GET
```

Maintaining a list of recent matches

- Goal: match packet sent by an IP address that has recently done something wrong
 - Example: drop all packets sent by a user that has previously tried to contact local port 30000
- The **recent** match one can dynamically create a list of IP addresses that match a rule and then match against these IPs in different ways later
- To add an IP source address of a packet that matches some rule use the following syntax
 - `-m recent --name "list_name" --set`
 - Example: put IP source address that contacts local port 30000 tcp in the list "bad_guys"
 - `iptables -A INPUT -p tcp -dport 30000 -m recent --name bad_guys --set -j DROP`
- To match the IP source address in bad_guys list use
 - `-m recent --name bad_guys --rcheck --seconds n`
 - After *n* seconds, the address is removed from the list
 - Example: check if a packet source address is in bad_guys list and if so don't forward it
 - `iptables -A FORWARD -m recent --name bad_guys --rcheck --seconds 10 -j DROP`
- **Homework:** combine recent and limit match to drop (for 2 minutes) all traffic generated by an IP address inside the LAN that has tried to connect remote ssh servers more than 10 times per minute

Packet owner match

- The **owner** match extension is used to match packets based on the user id or group id that “sends the packet”
 - i.e.: the uid or gid of the application that called the send()
- Syntax
 - -m owner --uid-owner root
 - -m owner --gid-owner net
- In older kernels it was possible to match also the PID of the process generating the outgoing packets
 - If you want to keep track of different applications use different users/groups per application
 - “apache” user, “ssh” user, etc etc..

Example

Assumptions

1. Two internet connections: 1) eth0 FAST; 2) eth1 SLOW
2. Policy routing configured so that: 1) mark=1 → eth0; 2) mark=2 → eth1

GOAL

Create two groups 1) fast, 2) slow

Create two users 1) uid: marco, gid: fast; 2) uid: lorenzo, gid: slow

Force marco and lorenzo to respectively use the fast and slow connection

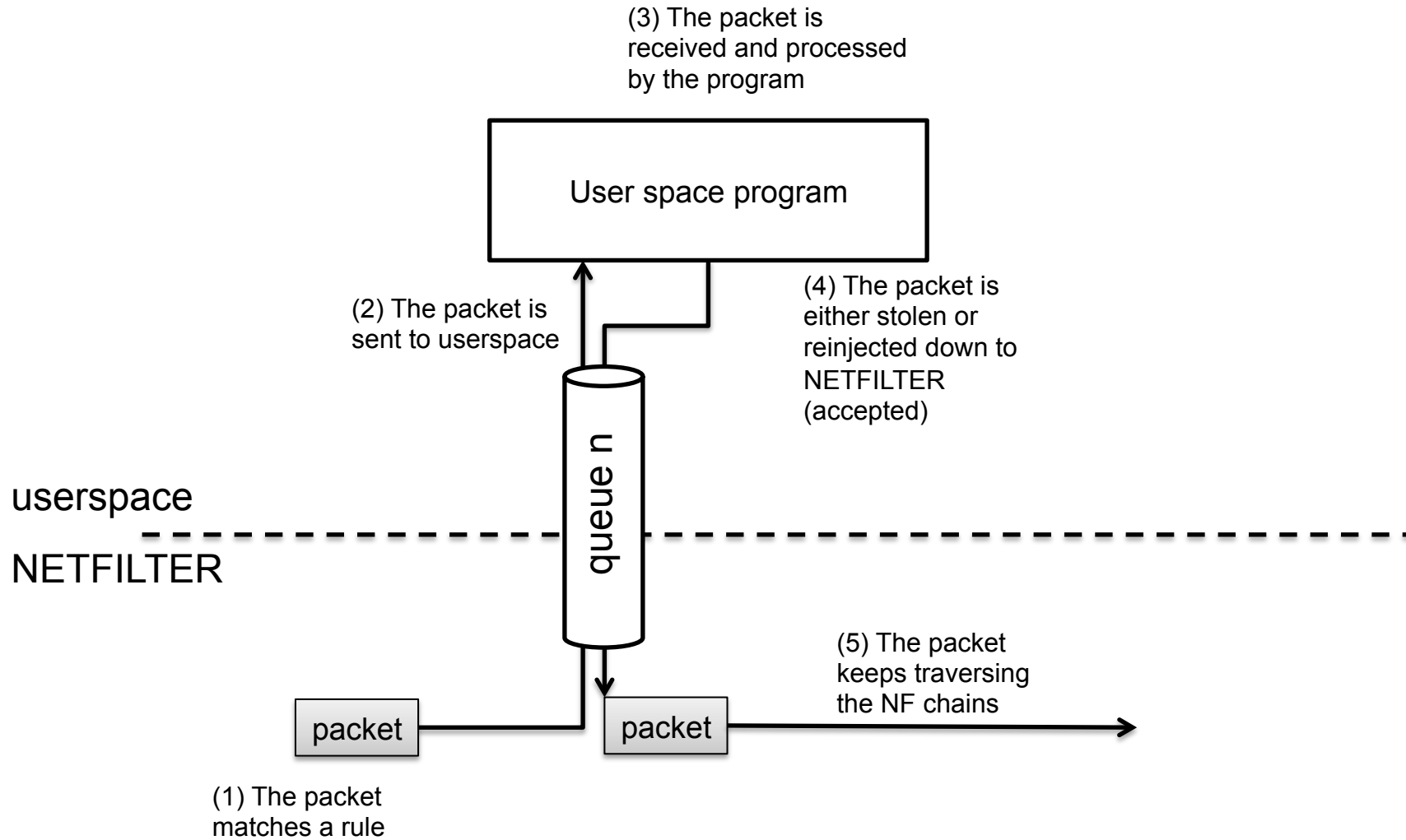
```
iptables -A OUTPUT -m owner --gid fast -j MARK --set-mark 1
```

```
iptables -A OUTPUT -m owner --gid slow -j MARK --set-mark 2
```

Queuing packets in userspace

- NETFILTER provides a target named **NFQUEUE** that enques packets to user space through a special type of sockets called NETLINK sockets
- Using “-j NFQUEUE” any matching packet will be sent over a netlink queue identified with a number “--queue *N*”
- In userspace, a program will be listening on the specified queue and will receive the entire matched packet
- The received packet can be processed in any ways and then re-injected in the NETFILTER chains (returning ACCEPT) or stolen (both DROP and STOLEN verdicts)
- **Example:** queue all packets sent by 10.0.0.2 to queue number 2
`iptables -A FORWARD -s 10.0.0.2 -j NFQUEUE --queue 2`

NFQUEUE structure



NFQUEUE configuration and usage

- There is no time (and this is not the right place) to play with NFQUEUE
- Just some directions
 - Project home:
http://www.netfilter.org/projects/libnetfilter_queue/index.html
 - Configure the kernel to support NFQUEUE
 - Install libnetfilter_queue and libnfnetlink (which the first one depends on)
 - In libnetfilter_queue tar there is an example main that show the basic NFQUEUE usage
http://www.netfilter.org/projects/libnetfilter_queue/doxygen/nfql_test_8c_source.html