

# CSIS0234B Computer and Communication Networks (Class B)

## Reading for Tutorial 8

### Firewall and NAT in Linux

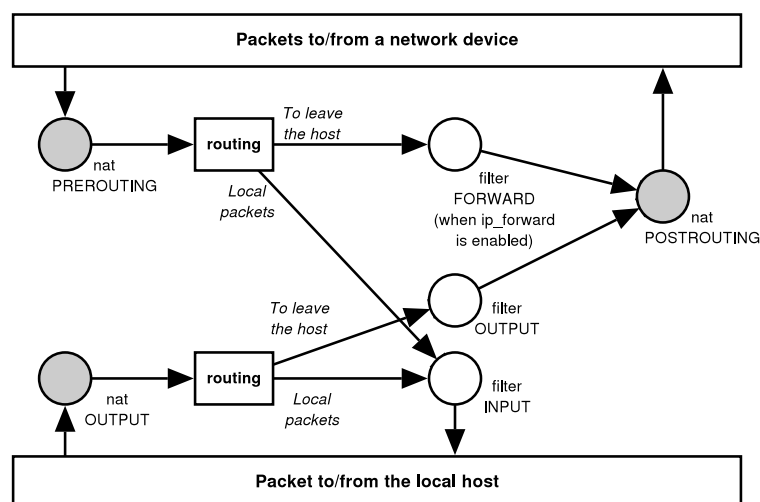
In the last tutorial we learn how to distribute a block of IP addresses within an organization. Unluckily, due to IP address shortage, organizations do not usually own a block of addresses. They typically have a single address allocated by their ISP when they make their Internet connection, and they want to use the address to serve multiple computers. To do this, a special “NAT router” is needed. While there are commercial product that does this, it is more flexible if we use a general purpose computer for NAT. We will see how Linux can serve the purpose.

The Linux NAT system is built on its packet filtering system—which controls whether to allow or reject each packet. Since it shares similar idea with firewall, we will examine both NAT and firewall in a single shot.

## 1. Overview of iptables

In the current 2.4 and 2.6 kernels, the packet filtering subsystem is based on a kernel module and a corresponding user program, both called **iptables**. It works by maintaining a set of filtering **rules** in the kernel. Each rule says if a packet **matches** some criteria, an **action** is taken. In iptables, rules live in different **chains**, which are used for different parts of the routing process. E.g., there is a separate chain for filtering packets that are incoming, outgoing and “in transit”. This differentiation makes it easier to write rules specific to each use case.

The chains live in kernel **tables** (as is implied by the name *iptables*). If a new set of functionalities (e.g., NAT) is needed, a table is added, which contains a set of **built-in chains**. We will discuss two tables: the default *filter* table, and the table *nat* for supporting NAT. The following figure describes when each chain in the tables are used. Note that each packet is processed by exactly one of the three chains of the *filter* table, immediately after the routing decision is made.



To avoid having to write the same set of rules in different chains, one can create **user-defined chains** in the tables. They can be thought of as “functions” in the world of Linux packet filter.

## 2. Linux firewall rules

When a packet is processed by a chain, the rules of the chain is tried one by one, until a rule

decides to terminate processing, or until all rules of a chain are used—when the **policy** (i.e., “default action”) of the chain is used.

Each filtering rule has some **criteria** to match: a source or destination IP address or port number, IP protocol, incoming or outgoing network interface, etc. (It is similar to `tcpdump` we learnt in tutorial 5, but less powerful and uses a different syntax.) If the criteria match, an **action** is taken. E.g., one might want to accept the packet or drop the packet. One might also call a user-defined chain or return from it.

These are specified with the `iptables` program. By default it operates on the `filter` table, and another table can be selected using `-t`. E.g., we can list all rules in the table using `iptables -L`. If you want to list the `nat` table instead, we use `iptables -t nat -L`. The `-L` option is called the **command** to `iptables`. Commands may take arguments. E.g., `-L` can take one argument, which is the chain to list—`iptables -L INPUT` lists only the `INPUT` chain. The most commonly used commands include:

- `-L [chain]` **List** the rules in the chain *chain*, by default list all chains.
- `-A chain rule-spec` **Append** *rule-spec* at the end of the rule. E.g., `iptables -A INPUT -i lo -s 127.0.0.0/8 -j ACCEPT` specifies that the rule `-i lo -s 127.0.0.0/8 -j ACCEPT` is to be appended to the `INPUT` chain.
- `-I chain [rule-num] rule-spec` **Insert** *rule-spec* at the beginning of *chain*, or after the *rule-num*-th rule if it is specified.
- `-D chain rule-spec` **Delete** chains with criteria *rule-spec* from *chain*.
- `-D chain rule-num` **Delete** the *rule-num*-th rule in *chain*.
- `-F chain` **Flush** *chain*, i.e., delete all rules in it.
- `-P chain` Set the **policy** (default action) of *chain*.
- `-N chain` Create a **new** *chain* as a user defined chain.
- `-X chain` Remove an empty, user defined chain *chain*.

The *rule-spec* is specified by a list of criterion and an action: all criteria must match for the action to be taken. Each of the criterion can be negated using a bang (!) character. Common criteria include:

- `-s address[/mask]` The packet has the source IP address in the network defined by *address* and *mask*, i.e., if the source address is bit-wised and with the *mask* the result is *address*. The *mask* can also be a simple number, in which case it is interpreted in the CIDR notation.
- `-d address[/mask]` Similar to `-s`, but checks the destination IP address.
- `-i [!] name` The packet comes from the interface *name*. It can only be used in `INPUT`, `FORWARD` and `PREROUTING` chains.
- `-o [!] name` The packet is heading to the interface *name*. It can only be used in `OUTPUT`, `FORWARD` and `POSTROUTING` chains.
- `-p [!] protocol` The packet has the IP protocol *protocol*. E.g., `tcp`, `udp`, `icmp`, etc.
- `--sport [!] port[:port]` The source port number (in case of `TCP` and `UDP`) of the packet is *port*, or

in the range *port:port*. This must be used together with *-p*.

`--dport [!]port[:port]`

Same as `--sport`, but checks destination ports.

Finally, we have a set of options for specifying the action to take, which always starts with the `-j` option. The commonly used options include:

`ACCEPT` Accept the packet and stop processing.

`DROP` Reject the packet and stop processing.

`REJECT` [`--reject-with type`]

Reject the packet, send ICMP response back to the initiator, and stop processing. The ICMP response is of type *type*, defaulting to `port-unreachable`.

*chain* Calls a user defined chain.

`RETURN` Returns from a user defined chain.

`LOG` [`--log-prefix prefix`]

Makes a log message in the log file. The message can have a *prefix* of up to 29 characters to help differentiating among different messages.

So the `-i lo -s 127.0.0.0/8 -j ACCEPT` rule that we see in the example above means “if the packet comes from the `lo` (loopback) interface and has source address in the network `127.0.0.0/8`, accept the packet and stop checking other rules in the same chain. As another example, the rule `-i ! lo -s 127.0.0.0/8 -j LOG --log-prefix 'Malicious packet: '` specifies that if a packet is not coming from the loopback interface but claims that it comes from `127.0.0.0/8`, make a log message with prefix “Malicious packet: ”. This is probably followed by a rule having the same criteria, with the action `DROP`. If this happens frequently enough, we might want to use a user-defined chain:

```
iptables -N log_drop_bad
iptables -A log_drop_bad -j LOG --log-prefix 'Malicious packet: '
iptables -A log_drop_bad -j DROP
iptables -A INPUT -i ! lo -s 127.0.0.0/8 -j log_drop_bad
```

### 3. NAT in Linux

NAT (Network Address Translation) allows the same IP address to be used for many computers. The idea is that the NAT router modifies the **source** IP address of a packet when an internal computer sends an outgoing packet to the NAT router. On the “return trip”, the incoming packet from the external network towards the NAT router modifies the **destination** address to the address of an internal computer. For this to work, the NAT router must watch for the use of new port numbers from other computers, allocate an unused port number of its own, and record that in memory so that all future packets from the same IP address and port number are modified similarly. Since port numbers are used, this can only be used for ICMP, UDP and TCP (it works for ICMP because ICMP has an ID which can be used like a port number).

All these work are done in the `iptables` module of the kernel. The module adds an `nat` table for NAT processing. Whenever a packet arrives from a previously unknown (source address, protocol, source port) triplet, the packet will pass through the chains in the `nat` table, in addition to the normal chains in the `filter` table. Like the `filter` table, its rules can have actions to allow or drop the packet. But it also has a few special actions for NAT processing:

SNAT `--to-source ipaddr[:port-port]`

Modify the source address of this packet to an IP address as specified in *ipaddr*, and modify all similar packets later similarly. The source port number might need to be modified as well if the router has already used that port. One may also specify a *port* range to be used for all such packets. The port number used is recorded internally, so that whenever a future packet arrives at that port with the destination address being that of the router, the destination IP address and port number is modified. To maximize the amount of information available for the filter rules, this action can only be taken in the `POSTROUTING` rule, just before the packet is sent to a network interface.

This is the classic use of NAT, called **source NAT**—the source address is to be modified in the first packet. The internal computers should then set the NAT router as their default router, with an **unroutable address** used as their IP address. After that the internal computers “seems” to be able to communicate with the outside world directly.

MASQUERADE `[--to-ports port-port]`

This is nearly the same as SNAT, although it is made to cater for the needs of users without a fixed IP address. Typically, such users will get an IP address dynamically from the ISP, which might be different for each connection. When this action is used instead of SNAT, the IP address of the outgoing interface of the NAT router is found and replaces the original source address. When that interface shuts down, the records of the NAT actions are flushed. For example, the following says all packets to `192.168.1.0/24` are to be masqueraded.

```
iptables -t nat -A POSTROUTING -d 192.168.0.24/24 -j MASQUERADE
```

DNAT `--to-destination ipaddr[-ipaddr][:port-port]`

This performs **destination NAT**, complementary to SNAT. The first packet, and all similar packets in the future, has the destination address modified to another IP address, as specified in the range *ipaddr*. The destination port number can also be modified, to one specified in the range *port*. In the reverse direction, the source address and port number is changed back to that of the original packet. Since the destination address is modified, this action must be taken **before** the routing decision is made, in the `PREROUTING` and `OUTPUT` chains.

Destination NAT is usually used for external hosts to access the service that is offered by an internal computer. To the external host, it is talking to a computer with the IP address of the NAT router. The NAT router provides the service using the internal host—by modifying the destination address to that of the internal computer and forward the packet to it. E.g.,

```
iptables -t nat -A PREROUTING --dport 80 \  
-j DNAT --to-destination 192.168.1.3
```

specifies that all HTTP (port 80) requests to the NAT router is to be forwarded to the internal computer `192.168.1.3`, without changing the destination port number. Since the destination IP address can be a range, this can also be used for **load-balancing**: for multiple internal computers to run the service.

Finally, some protocols like FTP and IRC use IP addresses of the end-points within the contents of the messages they send. The NAT mechanism would normally fail. The Linux kernel, however, deal with the problem by having a few modules to modify even the **contents** (rather than just the addresses) of the application layer messages (for the few application layer protocols that it supports), thus deals with the issue. This is enabled by loading the `ip_nat_ftp` and `ip_nat_irc` respectively.