CSIS0234B Computer and Communication Networks (Class B)
Reading for Tutorial 8
Firewall and NAT in Linux

In the last tutorial we see to allocate globally recognized IP addresses to LANs, and how routers can be statically configured to route IP packets correctly for those addresses. It works as long as you own a set of addresses. This is usually not the case for home and small business users, however. They typically have an address allocated by their ISP when they make their Internet connection, and they desire that the same address can be used to serve multiple computers. To do this, a special "NAT router" can be used. We will see how Linux can act as one.

The Linux NAT system is built on in its packet filtering system, which is used for building a firewall (i.e., to control which packets are allowed and what packets are rejected by the computer). So we will examine both NAT and firewall a single shot.

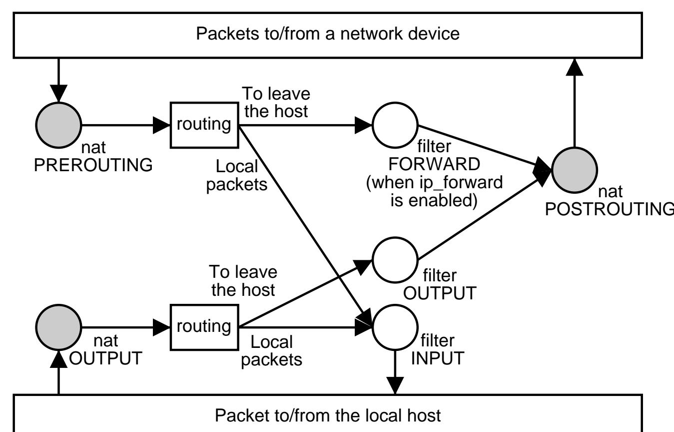
1. Firewall in Linux

Every minor version of the Linux kernel substantially updated the packet filtering subsystem—from the old 2.0 kernel when "`ipfwadm`" is used, to "`ipchains`" of the 2.2 kernels, to "`iptables`" of the 2.4 kernel. Conceptually they do the same thing: to decide whether to allow or reject IP packets based on a limited set of properties of the packets like their addresses and port numbers. But each replacement significantly simplify the configurations required, even for complicated environments. So we will look at the newest version.

The computers in our lab, with Redhat 7.3 installed, by default uses `ipchains`. To use `iptables`, one has to stop the `ipchains` service (using `service ipchains stop`), unload the `ipchains` module, load the `ip_tables` module, and start the `iptables` service instead.

`iptables` is based on the idea that the user configures some kernel **tables** to control the packet filter. Each functionality is implemented by a separate table. Currently, three tables are defined: regular firewalls use the `filter` table, NAT routers use an additional `nat` table, and the `mangle` table is used to specify more advanced rewriting of packets. We will focus on the first two.

Each table have a number of **built-in chains**, and may contain some **user-defined chains** as well. Different built-in chains are used depending on whether a packet is sent or received, and whether it is locally generated or comes from the network. The following diagram shows when each chain in the `filter` and `nat` table is used. Note that each packet goes through exactly one of the three built-in chains of the `filter` table, immediately after the routing decision is made.



Each chain has a number of **rules**, in the form "*if the packet matches this pattern, then do this action*". Since fire wallis primarily for controlling whether to accept or reject packets, the two primary actions are `ACCEPT` and `DROP`. An action can also call a user-defined chain, and can `RETURN` from such a chain. Each pattern try each rule one by one until one matches, and use the action of that matching rule. All built-in chains has a **policy**, which specifies what should happen if none of the rules matches. There is also a `LOG` action which allows a log message to be written to the log files, without terminating the matching process. See below for an example.

An action is triggered by a **pattern**. A pattern can match a source or destination IP address or port number, IP protocol name (tcp, udp or icmp), the name of the network interface (like `lo`, `eth0`, etc) from/to which the packet is received/sent, ICMP packet type (ping, ping-reply, etc), and others properties of the packet.

These are all configured using the `iptables` program. Using the program, you can select a table and a chain, and list/modify its content. E.g., you can display the current content of the `INPUT` chain of the `filter` table using this.

```
iptables -t filter -L INPUT
```

Here `-L` means list. If the table is not specified, it defaults to `filter`. If the chain is not specified, it shows all chains. By default it shows only the protocol, IP addresses and port to match. A `-v` option allows you to see the interface to match as well as the number of packets and bytes which is processed by each chain.

You can use `-A` and `-I` to append a rule to a chain and to insert a new rule at a particular position of the chain. We only illustrate the former, the latter is just the same except that it has a *rule number* (starting from 1) at the beginning, specifying where (i.e., after which rule) the new rule should be inserted. An example of the `-A` option:

```
iptable -A INPUT -i lo -s 127.0.0.0/8 -j ACCEPT
```

This specifies that every packet of the `lo` (loopback) interface claiming to be from address in the network 127.0.0.0–127.255.255.255 (the `/8` specifies the number of leading bits to match) should have the action `ACCEPT` (the `-j` means "jump", because the value can also be the name of a user-defined chain). Other possible flags include `-d` (destination IP address), `-p` (protocol), `-o` (output interface), `-sport` (TCP/UDP source port number), `-dport` (TCP/UDP destination port number), etc. In general, one can use `!` to negate the sense, i.e., match everything that is not a particular value. E.g., the following prevents anybody in the network to send you a packet pretending that it is sent from the local interface, and make a log message whenever it matches:

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

A rule can be removed from the chain using exactly the same syntax, replacing `-A` by `-D`. Alternatively, you can use `-D` and specify the rule number to delete (instead of the long pattern/action pair). To delete everything in a chain, a simpler way is to use `-F` (flush) instead.

Finally, you can use `-P` to specify the policy of a built-in chain. E.g., to make everything not matching any rule to be rejected in the `INPUT` chain, you can use

```
iptable -P INPUT DROP
```

2. NAT in Linux

NAT (Network Address Translation) allows the same IP address to be used for many computers. Whenever the right packet is passed to the NAT router, it modifies the IP addresses to the IP address of the NAT router itself, and change the port numbers to its own port number. See the lecture notes 7.18–7.19 for details. 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 UDP and TCP.

The `nat` table is used when new port numbers are found in a packet. Its chains decide whether the packet and future packets of the same IP address and port number should be processed normally (via the `ACCEPT` or `DROP` action), or should be modified (via an action described below).

There are two times at which modifications can be made. The first is when a packet is about to leave the router. This correspond to the `POSTROUTING` chain of `nat`. This is useful for the classical use of NAT as described above, called **source NAT**. For such use, the computers in the local network are set to have the NAT router as the default gateway. When a connection establishing packet is sent to the NAT (for forwarding), it is processed normally, but just when the packet is about to be sent to the network device, a new NAT entry is created, and from then on all the packets with that IP addresses and port numbers are modified according to the entry (modifying the source IP address and port number). The action to activitate the feature are `MASQUERADE` and `SNAT`, which are basically the same except that in `MASQUERADE`, whenever an interface is brought down, the NAT table is cleared. This is useful for those hosts with dynamic IP addresses assigned by their ISP, since once the connection to ISP hangs up and is redialed, a new IP address is obtained, and the old connection are unusable. As an example of using `MASQUERADE`, the following specifies that NAT processing is done for every packet from the local network `192.168.1.0/24` to `eth0`:

```
iptable -t nat -A POSTROUTING -s 192.168.1.0/24 -o eth0 \
  -j MASQUERADE
```

The other time at which modification can be done is when a packet enters a router, either because it is locally generated or because it is received from a network device. This correspond to the `OUTPUT` and `PREROUTING` chains of `nat` respectively (although the `OUTPUT` chain in the `nat` table only starts working from Linux kernel 2.4.19, a bit newer than the Redhat 7.3 installed in the lab). They are useful when the NAT router is used for computers in the Internet to access a service by using the IP address of the NAT router, although the service is actually implemented in one of the computers within the local network (and is thus normally invisible from outside). So although the same IP address is used to implement perhaps a large number of services, the CPU loading is spread to computers in its local network. This is called **destination NAT**. Whenever a packet arrives and is to be routed, it can modify the destination address (using `DNAT`) so that it is routed to another computer within the local network. It also allows the port number to be changed in the process. As an example, the following establishes a web server at the computer with local IP address `192.168.1.3`, although accessible through the real IP address of the NAT:

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

In fact, it is even possible to use a range of IP addresses as the `-to-destination` option, thus allowing a group of machines to share the load of a very busy server.

Finally, some protocols like FTP and IRC uses IP addresses of the end-points within the contents of the messages they send. The NAT mechanism would normally fail completely for them. The Linux kernel, however, assists the situation by having a few modules to modify even the **content** (rather than just the address) of the application layer messages (for the few application layer protocols that it supports), thus deals with the issue (it also need to modify some of its own table. E.g., ftp will make a reverse connection, so the NAT must be prepared for that). This is enabled by loading the `ip_nat_ftp` and `ip_nat_irc` respectively.