Lecture 18: Packet Filtering Firewalls (Linux)

Lecture Notes on "Introduction to Computer Security"

by Avi Kak (kak@purdue.edu)

April 5, 2007

Goals:

- Packet-filtering vs. proxy-server firewalls
- The four iptables supported by the Linux kernel: **filter**, **nat**, **mangle**, and **raw**
- Creating and installing new firewall rules
- Structure of the **filter** table
- Connection tracking and extension modules
- Designing your own filtering firewall

Firewalls in General

- Two primary types of firewalls are
  - packet filtering firewalls
  - proxy-server firewalls

  Sometimes both are employed to protect a network. A single computer may serve both roles.

- A proxy-server firewall handles various network services itself rather then passing them straight through. What exactly that means will be explained in the lecture on proxy server firewalls.

- In Linux, a packet filtering firewall is configured with the iptables modules.

- To use iptables for a packet filtering firewall, you need a Linux kernel which has the **netfilter infrastructure** in it: **netfilter** is a general framework inside the Linux kernel which other things (such as the **iptables module**) can plug into. This means you need kernel 2.3.15 or beyond, and answer 'Y' to **CON-**

FIG_NETFILTER in the kernel configuration. (The tool **iptables** talks to the kernel and tells it what packets to filter.)

- The **iptables** tool inserts and deletes rules from the kernel's packet filtering table. Ordinarily, these rules created by the **iptables** command would be lost on reboot. However, you can make the rules permanent with the commands **iptables-save** and **iptables-restore** scripts. The other way is to put the commands required to set up your rules in an initialization script, which is currently the case with my personal laptop.

- Note that **iptables** replaces the older **ipfwadm** and **ipchains**. However, **iptables** based firewalls are very similar to the previous **ipchains** based firewalls.

- Rusty Russell wrote **iptables**. He is also the author of **ipchains** that was incorporated in version 2.2 of the kernel and that was replaced by **iptables** in version 2.4.

The Four Tables Maintained by the Linux Kernel for Packet Processing

- Linux kernel uses the following **four tables**, each consisting of rule chains, for processing the incoming and outgoing packets:

  - the **filter** table

  - the **nat** table

  - the **mangle** table

  - the **raw** table

- Each table consists of **chains of rules**.

- Each packet is subject to each of the rules in a table and the fate of the packet is decided by the first matching rule.

- The **filter** table contains at least three rule chains: **INPUT** for processing all incoming packets, **OUTPUT** for processing all outgoing packets, and **FORWARD** for processing all packets being routed through the machine.

- The **INPUT**, **OUTPUT**, and **FORWARD** chains of the **filter** table are also referred to as the **built-in chains** since they cannot be deleted (unlike the user-defined chains we will talk about later).

- The **nat** table is consulted when a packet that creates a new connection is encountered.

- The **nat** table also consists of three built-in chains: **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.

- The **mangle** table is used for specialized packet alteration.

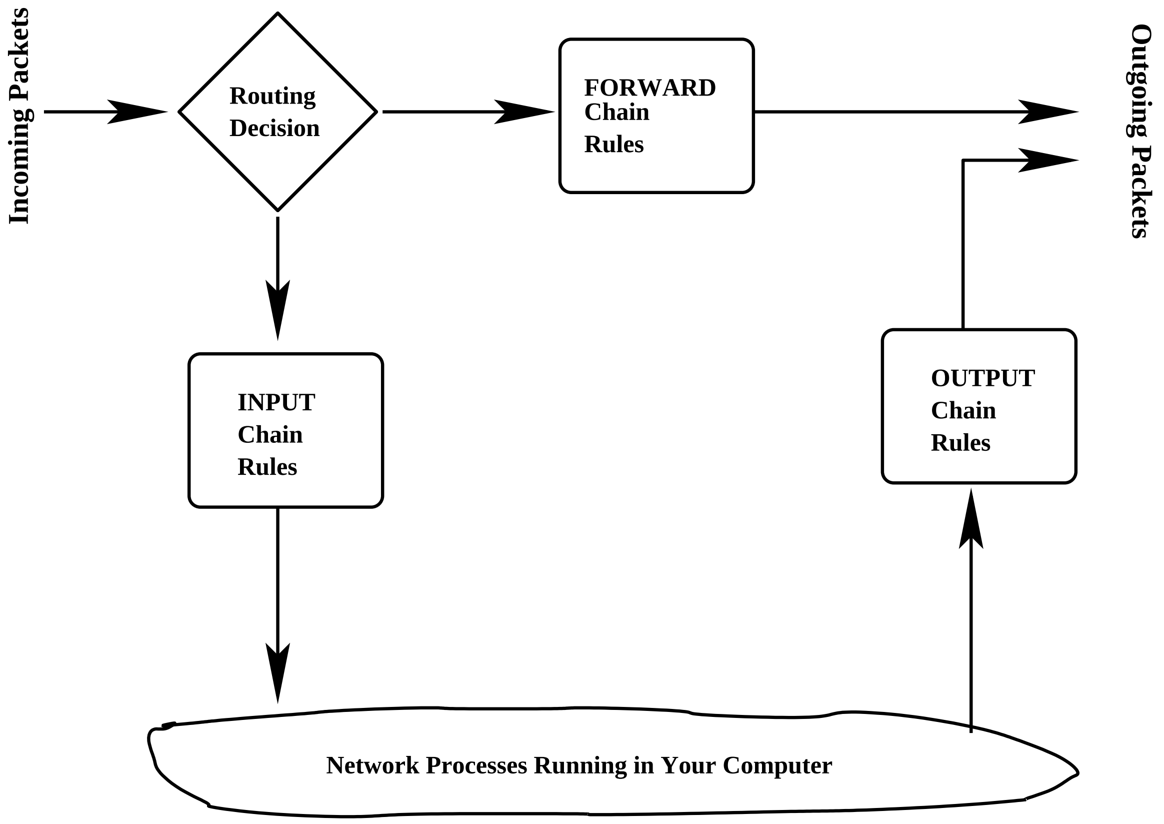
- The **mangle** table has five rule chains: **PREROUTING** for altering incoming packets before routing, **OUTPUT** for altering locally generated outgoing packets before routing, **INPUT** for altering packets coming into the machine itself, **FORWARD** for altering packets being routed through the machine, and **POSTROUT-ING** for altering packets as they are about to go out.

- We will focus most of our attention on the **filter** table since that is the most important table for firewall security.

How Packets are Processed by the `filter` Table

- As mentioned already, the **filter** table contains the following built-in rule chains: **INPUT**, **OUTPUT**, and **FORWARD**.

- The figure below shows how a packet may be subject to these rule chains in the Linux 2.3 kernel.



- When a packet comes in (say, through the Ethernet card) the kernel first looks at the destination of the packet. This step is labeled 'routing' in the figure.

- If the routing decision is that the packet is intended for the machine in which the packet is being processed, the packet passes downwards in the diagram to the **INPUT** chain.

- If the incoming packet is destined for another network interface on the machine, then the packet goes rightward in our diagram to the **FORWARD** chain. If **ACCEPT**ed by the **FORWARD** chain, the packet is sent to the other interface. (But note that if the kernel does not have forwarding enabled or if the kernel does not know how to forward the packet, the packet would be dropped.)

- If a program running on the computer wants to send a packet out of the machine, the packet must traverse through the **OUTPUT** chain of rules. If it is **ACCEPT**ed by any of the rules, it is sent to whatever interface the packet is intended for.

- Each rule in a chain will, in general, examine the packet header. If the condition part of the rule matches the packet header, the action specified by the rule is taken. Otherwise, the packet moves on to the next rule.

- If a packet reaches the end of a chain, then the Linux kernel looks at what is known as the **chain policy** to determine the fate of the packet. **In a security-conscious system, this policy usually tells the kernel to DROP the packet.**

To See if `iptables` are Installed and Running

- Execute as root

  ```
  lsmod | grep ip
  ```

  On my laptop, this returns

  ```
  ipt_REJECT              10689  1
  ipt_state                5953  1
  ip_conntrack            46085  1 ipt_state
  iptable_filter           6977  1
  ip_tables               22721  3 ipt_REJECT,ipt_state,iptab
  ipv6                   242657  12
  ```

  If you see the **`ipchains`** module running, as you might with older installations of Linux, unload that module. You can do that by first stopping **`ipchains`**:

  ```
  /etc/init.d/ipchains stop
  ```

  followed by

  ```
  modprobe -r ipchains
  ```

- Another way to see if `iptables` is installed and running, execute as root

  ```
  iptables -L
  ```

  On my Linux laptop, this command line returns

  ```
  Chain INPUT (policy ACCEPT)
  target               prot opt source     destination
  RH-Firewall-1-INPUT  all  --  anywhere   anywhere

  Chain FORWARD (policy ACCEPT)
  target               prot opt source     destination
  RH-Firewall-1-INPUT  all  --  anywhere   anywhere

  Chain OUTPUT (policy ACCEPT)
  target     prot opt source               destination

  Chain RH-Firewall-1-INPUT (2 references)
  target     prot   opt   source           destination
  ACCEPT     all    --    anywhere         anywhere
  ACCEPT     icmp   --    anywhere         anywhere    icmp any
  ACCEPT     ipv6-crypt --  anywhere       anywhere
  ACCEPT     ipv6-auth  --  anywhere       anywhere
  ACCEPT     udp    --    anywhere         224.0.0.251 udp dpt:5353
  ACCEPT     udp    --    anywhere         anywhere    udp dpt:ipp
  ACCEPT     all    --    anywhere         anywhere    state RELATED,ESTABLISHED
  REJECT     all    --    anywhere         anywhere    reject-with icmp-host-prohibite
  ```

- The invocation `iptables -L` shows only the **filter** table. In general, if you want to see the rules of a particular table, you would call

```
iptables -t filter -L              (to see the filter table)
iptables -t nat    -L              (to see the nat table)
iptables -t mangle -L              (to see the mangle table)
iptables -t raw    -L              (to see the raw table)
```

Note that these are the only four tables recognized by the kernel. (Unlike user-defined chains in the tables, there are no user-defined tables.)

- For the **filter** table shown above, note the policy shown for each built-in chain right next to the name of the chain. As mentioned earlier, only built-in chains have policies.

- For each rule, the keyword **ACCEPT** means to let the packet through, the keyword **DROP** means to drop the packet, **QUEUE** means to pass the packet to userspace (if supported by the kernel), **RETURN** means to stop traversing this chain of rules 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 determined by the chain policy determines the fate of the packet. Note that a policy is only associated with a built-in chain.

Structure of the `filter` Table

- First, you can supply the **`iptables`** command with additional options so that the displayed table is visually easier to understand. For example,

  ```
  iptables -n -L
  ```

  will use the numerical format (the decimal quad notation) for the source and destination addresses in the **`filter`** table. If you want the **`filter`** table to be displayed with line numbers for the individual rows of the table, the following command does that:

  ```
  iptables -L --line-numbers
  ```

- The following command with the 'verbose' option turned on refers to protocols and ports by name:

  ```
  iptables -L -v --line-numbers
  ```

  On my laptop, this command returns

```
1       Chain INPUT (policy ACCEPT)
2       target              prot opt  source    destination
3       RH-Firewall-1-INPUT  all  --  0.0.0.0/0   0.0.0.0/0

4       Chain FORWARD (policy ACCEPT)
5       target               prot opt source    destination
6       RH-Firewall-1-INPUT  all  --  0.0.0.0/0   0.0.0.0/0
```

```
7       Chain OUTPUT (policy ACCEPT)
8       target               prot opt source    destination

9       Chain RH-Firewall-1-INPUT (2 references)
10      target     prot opt source         destination
11      ACCEPT     all  --  0.0.0.0/0      0.0.0.0/0
12      ACCEPT     icmp --  0.0.0.0/0      0.0.0.0/0       icmp type 255
13      ACCEPT     esp  --  0.0.0.0/0      0.0.0.0/0
14      ACCEPT     ah   --  0.0.0.0/0      0.0.0.0/0
15      ACCEPT     udp  --  0.0.0.0/0      224.0.0.251     udp dpt:5353
16      ACCEPT     udp  --  0.0.0.0/0      0.0.0.0/0       udp dpt:631
17      ACCEPT     all  --  0.0.0.0/0      0.0.0.0/0       state RELATED,ESTABLISHED
18      REJECT     all  --  0.0.0.0/0      0.0.0.0/0       reject-with icmp-host-prohibit
```

- In the output shown above, note that the last column, with no heading, can mention the port(s) to which the rule applies. The acronym `dpt` in the last column means **destination port**. When the last column entry is empty, that means no applicable ports when the second column mentions `all`. On the other hand, when the second column mentions a specific service, then the last column must mention the port specifically if it is a user-defined service. If it is a standard service, the system can figure out the ports from the entries in `/etc/services`.

- Note the three rule chains in the `filter` table: **INPUT**, **FORWARD**, and **OUTPUT** in lines 1, 4, and 7, respectively. Most importantly, note how the **INPUT** chain jumps to the user-defined chain **RH-Firewall-1-INPUT** in line 3. The built-in **FORWARD** chain also jumps to the same user-defined chain in line 6.

- About the **FORWARD** chain, note that packet forwarding only occurs when the machine is configured as a router. (For IP packet forwarding to work, you have also have to change the value of `net.ipv4.ip_forward` to 1 in the `/etc/sysctl.conf file`.)

- Lines 1, 4, and 7 declare the **policy** to **ACCEPT**. As mentioned previously, the policy sets the fate of a packet does not get trapped by any of the rules in a chain.

- Since both the built-in **INPUT** and the built-in **FORWARD** chains jump to the user-defined **RH-Firewall-1-INPUT** chain, let's look at the rules in this user-defined chain in some detail. The rule in line 11 is

```
 target    prot opt source     destination    (ports)
---------------------------------------------------------
ACCEPT     all  --  0.0.0.0/0   0.0.0.0/0       ---
```

  The `IP/port` address `0.0.0.0/0` means all addresses. Since no ports are mentioned, this means that this rule only applies to packets generated by users on the local system. (That is, this rule allows the loopback driver to work.) Therefore, with this rule, you can request any service from your local system without the packet being denied.

- Let's now examine the entry in line 12 for the user-defined chain **RH-Firewall-1-INPUT**:

```
 target    prot opt source     destination    (ports)
---------------------------------------------------------
 ACCEPT    icmp --  0.0.0.0/0  0.0.0.0/0    icmp type 255
```

As mentioned in Lecture16, ICMP messages are used for a basic kind of error reporting between host to host, or host to gateway. (Between gateway to gateway, a protocol called Gateway to Gateway protocol (GGP) should normally be used for error reporting.) (The headers of the ICMP messages are rather complicated, and differ a little bit from message to message. ICMP protocol lives on the same level as the IP protocol in a sense. As also mentioned in Lecture 18, ICMP does use the IP protocol as if it (that is, ICMP) were a higher level protocol, but at the same time not. ICMP is an integral part of IP, and ICMP must be implemented in every IP implementation. The three types of commonly used ICMP headers are `type 0`, `type 8`, and tt type 11. Type 8 service, which allows your computer to accept 'echo reply messages' makes it possible for people to ping your computer to see if it is available. Whereas the echo request is a type 8 ICMP message, an echo reply is a type 0 ICMP message. When a host receives a type 8, it replies with a type 0. Type 11 service relates to packets whose 'time to live' (TTL) was exceeded in transit and for which you as sender is accepting a 'Time Exceeded' message that is being returned to you. You need to accept type 11 ICMP protocol messages if you want to use the 'traceroute' command to find broken routes to hosts you want to reach. **See Slides 26 and 27 for additional ICMP types.**)

- Let's now examine the **OUTPUT** chain in lines 7 and 8. There are no rules in this chain. Any, for all outbound packets, the policy associated with the **OUTPUT** chain will be used. This policy says `ACCEPT`, implying that all outbound packets will be sent directly, without further examination, to their intended destinations.

- Note that in the table shown above, the first column is referred to as the **target**. The target is the action part of a rule. For the `filter` table, a target can be one of the following:

  1. **name of a user-defined chain to jump to** (as in lines 3 and 6 on Slide 13)

  2. **ACCEPT** (accept and deliver in the normal way)

  3. **DROP** (drop the packet completely)

  4. **REJECT** (This has the same effect as **DROP**, except that the sender is sent an ICMP 'port unreachable' error message. Note that the ICMP error message is not sent if (see RFC 1122):

     – The packet being filtered was an ICMP error message in the first place, or some unknown ICMP type.

     – The packet being filtered was a non-head fragment.

     – We've sent too many ICMP error messages to that destination recently (see `/proc/sys/net/ipv4/icmp_ratelimit`).)

     REJECT also takes a '–reject-with' optional argument which alters the reply packet used: see the manpage.)

5. **REDIRECT** (redirect the packet to a new location; used with nat)

6. **RETURN** (return from this chain to the chain that called it and continue examining rules in the calling chain where you left off When RETURN target is encountered in a built-in chain, the policy of the chain is executed.)

- In other words, when a packet header matches a rule, the action taken on the packet is referred to as the target. Some people do not include the jump specification (the first item above) in the targets listed in a ruleset. Such people say that when a packet header matches a rule, either a jump or a target instruction will be carried out ordinarily.

Structure of the `nat` Table

- Let's now examine the output produced by the command line

```
iptables -t nat -n -L
```

  we get

```
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

- The `nat` table is used only for translating either the packet's source address field or its destination address field.

- NAT (which stands for Network Address Translation) allows a host or several hosts to share the same IP address. For example, let's say we have a local network consisting of 5-10 clients. We set their default gateways to point through the NAT server.

The NAT server receives the packet, rewrites the source and/or destination address and then recalculates the checksum of the packet.

- Only the first packet in a stream of packets hits this table. After that, the rest of the packets in the stream will have this network address translation carried out on them automatically.

- The 'targets' for the nat table are

```
DNAT
SNAT
MASQUERADE
REDIRECT
```

- The **DNAT** target is mainly used in cases where you have a **single** public IP for a local network in which different machines are being used for different servers. When a remote client wants to make a connection with a local server using the publicly available IP address, you'd want your firewall to rewrite the destination IP address on those packets to the local address of the machine where the server actually resides.

- **SNAT** is mainly used for changing the source address of packets. Using the same example as above, when a server residing on one of the local machines responds back to the client, initially the packets emanating from the server will bear the source address of the local machine that houses the server. But as these packets pass through the firewall, you'd want to change the source IP address in these packets to the single public IP address for the local network.

- The **MASQUERADE** target is used in exactly the same way as **SNAT**, but the **MASQUERADE** target takes a little bit more overhead to compute. Whereas **SNAT** will substitute a single previously specified IP address for the source address in the outgoing packets, **MASQUERADE** can substitute a DHCP IP address (that may vary from connection to connection).

- Note that in the output of `iptables -t nat -n -L` shown above, we did not have any targets in the `nat` table. That is because my laptop is not configured to serve as a router.

Structure of the `mangle` Table

- The mangle table is used for changing fields such as the TOS field (Type of Service) or the TTL (Time to Live), etc., in a packet.

  On my Linux laptop, the command

  ```
  iptables -t mangle -n -L
  ```

  returns

  ```
  Chain PREROUTING (policy ACCEPT)
  target     prot  opt  source  destination

  Chain INPUT (policy ACCEPT)
  target     prot  opt  source  destination

  Chain FORWARD (policy ACCEPT)
  target     prot  opt  source  destination

  Chain OUTPUT (policy ACCEPT)
  target     prot  opt  source  destination

  Chain POSTROUTING (policy ACCEPT)
  target     prot  opt  source  destination
  ```

  A mangle table can only use the following targets:

1. **TOS** (Used to change the TOS field in a packet. Not understood by all the routers.)

2. **TTL** (The TTL target is used to change the TTL (Time To Live) field of the packet.)

3. **MARK** (This target is used to give a special mark value to the packet. Such marks are recognized by the iproute2 program for routing decisions.)

4. **SECMARK** (This target sets up a security-related mark in the packet. Such marks can be used by SELinux fine-grained security processing of the packets.)

5. **CONNSECMARK** (This target places a connection-level mark on a packet for security processing.)

How the Tables are Actually Created

- The iptables are created by the **iptables** command that is run as root with different options. To see all the option, say

```
iptables -h
```

- Here are some other optional flags for the iptables command and a brief statement of what is achieved by each flag:

```
iptables -N chainName          (Create a new user-defined chain)

iptables -X chainName          (Delete a user-defined chain; must have
                                been previoiusly emptied of rules by
                                either the '-D' flag or the '-F' flag.)

iptables -P chainName ....     (Change the policy for a built-in chain)

iptables -L chainName          (List the rules in a chain.  If no
                                chain specified, it lists rules in all
                                the chains in the filter table.  Without
                                the '-t' flag, the filter table is the
                                default for the '-L' flag.)

iptables -F chainName          (Flush the rules out of a chain)
                               (When no chain-name is supplied as the
                                argument to '-F', all chains are flushed.)

iptables -Z chainName          (Zero the packet and byte counters
                                on all rules in the chain)

iptables -A chainName ....     (Append a new rule to the chain)
iptables -I chainName pos .... (Insert a new rule at position 'pos'
                                in the specified chain)
                               (See an example of rule insertion in
                                my 'experiment' file)

iptables -R chainName ....     (Replace a rule at some position in
                                the specified chain)

iptables -D chainName ....     (Delete a rule at some position in
                                the specified chain, or the first that
                                matches)
                               (See my 'experiment' file for the
                                two different ways in which the '-D'
                                flag can be used)

iptables -P chainName target   (Specify a target policy for the chain.
                                This can only be done for built-in
                                chains.)
```

- Note that `chainName` in all the invocations of iptables listed above will be either the built-ins **INPUT**, **OUTPUT**, or **FORWARD**, or **the name of a user-defined chain**.

- After the first level flags shown above that name a chain, if this flag calls for a new rule to be specified (such as for '-A' flag) you can have additional flags that specify the state of the packet that must be true for the rule to apply and specify the action part of the rule. We say that these additional flags describe the **filtering specifications** for each rule.

- Here are the rule specification flags:

```
-p  args
          for specifying the protocol (tcp, udp,
          icmp, etc)  You can also specify a protocol
          by number if you know the numeric protocol
          values for IP.

-s  args
          for specifying source address(es)

--sport args
          for specifying source port(s)

-d  args
          for specifying destination address(es)

--dport args
          for specifying destination port(s)
          (For the port specifications, you can supply
          a port argument by name, as by 'www', as
          listed in /etc/services.)

--icmp-type typemane
        (  for spcifying the type of ICMP packet.  The
           icmp type names can be found by the comamnd

              iptables -p icmp --help

          it returns the following for the icmp types

           Valid ICMP Types:
                  any
                  echo-reply (pong)        (type 0)
                  destination-unreachable  (type 3)
                     network-unreachable      (code 0)
                     host-unreachable         (code 1)
                     protocol-unreachable     (code 2)
                     port-unreachable         (code 3)
                     fragmentation-needed     (code 4)
                     source-route-failed      (code 5)
                     network-unknown          (code 6)
                     host-unknown             (code 7)
                     network-prohibited       (code 8)
                     host-prohibited          (code 9)
                     TOS-network-unreachable  (code 10)
                     TOS-host-unreachable     (code 11)
                     communication-prohibited (code 12)
                     host-precedence-violation
                     precedence-cutoff
                  source-quench            (type 4)
                  redirect                 (type 5)
                     network-redirect
                     host-redirect
                     TOS-network-redirect
                     TOS-host-redirect
                  echo-request (ping)      (type 8)
                  router-advertisement     (type 9)
                  router-solicitation      (type 10)
                  time-exceeded (ttl-exceeded)(type 11)
                     ttl-zero-during-transit    (code 0)
                     ttl-zero-during-reassembly (code 1)
                  parameter-problem        (type 12)
                     ip-header-bad
                     required-option-missing
                  timestamp-request        (type 13)
                  timestamp-reply          (type 14)
                  address-mask-request     (type 17)
                  address-mask-reply       (type 18)  )

-j  args
          the name of the target to execute when
          the rule matches; 'j' stands for 'jump to'

-i  args
          for naming the input interface (when an
          interface is not named, that means all
          interfaces)

-o  args
          for specifying an output interface

          (Note that an interface is the physical
           device a packet came in on or is going
           out on. You can use the ifconfig command
           to see which interfaces are up.)
```

```
            (Also note that only the packets traversing the
             FORWARD chain have both input and output
             interfaces.)

            (It is legal to specify an interface that
             currently does not exist.  Obvously, the
             rule would not match until the interface
             comes up.)

            (When the argument for interface is followed
             by '+', as in 'eth+', that means all
             interfaces whose names begin with the
             string 'eth'.)

            (So an interface specified as
                 -i ! eth+
             means none of the ethernet interfaces.)

-f          (For specifying that a packet is
             a second or a further fragment.  As mentioned
             in Lecture 16 notes, sometimes, in order
             to meet the en-route or destination
             hardware constraints, a packet may have
             to be fragmented and sent as multiple
             packets.  This can create a problem for
             packet-level filtering since only the first
             fragment packet may carry all of the headers,
             meaning the IP header and the enveloped
             higher-level protocol header such as the TCP,
             or UDP, etc., header.  The subsequent fragments
             may only carry the IP header and not mention
             the higher level protocol headers.  Obviously,
             such packets cannot be processed by rules that
             mention higher level protocols. Thus a rule
             that carries the specification
                  '-p TCP --sport www'
             will never match a fragment (other than the
             first fragment). Neither will the opposite
             rule
                  '-p TCP --sport ! www'

             However, you can specify a rule specifically
             for the second and further fragments, using
             the '-f' flag. It is also legal to specify
```

that a rule does not apply to second and further fragments, by preceding the '-f' with ' ! '.

Usually it is regarded as safe to let second and further fragments through, since filtering will effect the first fragment, and thus prevent reassembly on the target host; however, bugs have been known to allow crashing of machines simply by sending fragments.)

(Note that the '-f' flag does not take any arguments.)

--syn (To indicate that this rule is meant for a syn packet. It is sometimes useful to allow TCP connections in one direction, but not in the other. For example, you might want to allow connections to an external WWW server, but not connections from that server. The naive approach would be to block TCP packets coming from the server. Unfortunately, a TCP connection between a client and a server require packets going in both directions. The solution is to block only the packets used to request a connection. These packets are called SYN packets (ok, technically they're packets with the SYN flag set, and the RST and ACK flags cleared, but we call them SYN packets for short). By disallowing only these packets, we can stop attempted connections in their tracks. The '--syn' flag is used for this: it is only valid for rules which specify TCP as their protocol. For example, to specify TCP connection attempts from 192.168.1.1:

```
-p TCP -s 192.168.1.1 --syn
```

This flag can be inverted by preceding it with a '!', which means every packet other than the connection initiation.)

```
-m match       (This is referred to as a rule seeking an
                'extended match'. This may load extensions
                to iptables.)

-n             (This forces the output produced by the
                '-L' flag to show numeric values for
                the IP addresses and ports.)
```

- Many rule specification flags (such as '-p', '-s', '-d', '-f' '–syn', etc.) can have their arguments preceded by '!' (that is pronounced 'not') to match values not equal to the ones given. This is referred to as **specification by inversion**. For example, to indicate all sources addresses but a specific address, you would have

```
-s ! ip_address
```

- For the '-f' option flags, the inversion is done by placing '!' before the flag, as in

```
! -f
```

  The rule containing the above can only be matched with the first fragment of a fragmented packet.

- Also note that the '–syn' second-level option is a shorthand for

  ```
  --tcp-flags SYN,RST,ACK SYN
  ```

  where `--tcp-flags` is an example of a TCP extension flag. Note that '-d', and '-s' are also TCP extension flags. These flags work only when the argument for the protocol flag '-p' is 'tcp'.

- The source ('-s', '–source' or '–src') and destination ('-d', '–destination' or '–dst') IP addresses can be specified in four ways:

  1. The most common way is to use the full name, such as `localhost` or `www.linuxhq.com`.

  2. The second way is to specify the IP address such as 127.0.0.1.

  3. The third way allow specification of a group of IP addresses with the notation 199.95.207.0/24 where the number after the forward slash indicates the number of leftmost bits in the 32 bit address that must remain fixed. Therefore, 199.95.207.0/24 means all IP addresses between 199.95.207.0 and 199.95.207.255.

  4. The fourth way uses the net mask directly to specify a group

of IP addresses. What was accomplished by 199.95.207.0/24 above is now accomplished by 199.95.207.0/255.255.255.0.

- If nothing comes after the forward slash in the prefix notation for an IP address range, the default of /32 (which is the same as writing down the net mask as /255.255.255.255) is assumed. Both of these imply that all 32 bit must match, implying that only one IP address can be matched. Obviously, the opposite of the default /32 is /0. This means all 32 address bits can be anything. Therefore, /0 means every IP address. The same is meant by the specifying the IP address range as 0/0 as in

  ```
  iptables -A INPUT -s 0/0 -j DROP
  ```

  which will cause all incoming packets to be dropped. But note that `-s 0/0` is redundant here because not specifying the '-s' flag is the same as specifying '-s 0/0' since the former means all possible IP addresses.

Connection Tracking by `iptables` and the Extension Modules

- A modern iptables-based firewall understands the notion of a stream. This is done by what is referred to as connection tracking.

- Connection tracking is based on the notion of '**the state of a packet**'.

- If a packet is the first that the firewall sees or knows about, it is considered to be in state `NEW` (consider, for example, the SYN packet in a TCP connection), or if it is part of an already established connection or stream that the firewall knows about, it is considered to be in state `ESTABLISHED`.

- States are known through the connection tracking system, **which keeps track of all the sessions.**

- *It is the connection-tracking made possible by Rule 17 on Slide* 14 *that when I make a connection with a remote host* *such as* `www.nyt.com` *that I am able to receive all the incoming packets. Rule 17 tells the kernel that the incoming packets are of state* **ESTABLISHED**, *meaning that they belong to a connection that was establihsed and accepted previously.*

- Connection tracking is also used by the **nat** table and by its **MASQUERADE** target in the tables.

- Let's now talk about extension modules since it is one of those extensions to **iptables** that makes it possible to carry out connection tracking.

- When invoking `iptables`, an extension module can be loaded into the kernel for additional match options for the rules. **An extension module is specified by the '-m' option.**

- A most useful extension module is 'state'. This extension tries to interpret the connection-tracking analysis produced by the `ip_conntrack` module.

- As to how exactly the interpretation of the results on a packet produced by the `ip_conntrack` module should be carried is specified by the additional '`--state`' option supplied to the '`-m state`' extension module.

- The '`--state`' option supplies a comma-separated list of states of the packet that must be found to be true for the rule to apply, and as before, the '!' flag indicates not to match those states. These states that can be supplied as arguments to the '`--state`' option are:

```
NEW               (A packet which creates a new connection.)

ESTABLISHED       (A packet which belongs to an existing
                   connection (i.e., a reply packet, or
                   outgoing packet on a connection which
                   has seen replies).)

RELATED           (A packet which is related to, but not
                   part of, an existing connection, such as
                   an ICMP error, or (with the FTP module
                   inserted), a packet establishing an ftp
                   data connection.)

INVALID           (A packet which could not be identified
                   for some reason: this includes running
                   out of memory and ICMP errors which
                   don't correspond to any known connection.
                   Generally these packets should be dropped.)
```

- An example of a rule that uses the '`-m state`' extension for stating the rule matching conditions:

```
iptables -A FORWARD -i ppp0 -m state ! --state NEW -j DROP
```

This says to append to the FORWARD chain a rule that applies to all packets being forwarded through the ppp0 interface. If such a packet is NOT requesting a new connection, it should be dropped.

- Another extension module is the 'mac' module that can be used for matching an incoming packet's source Ethernet (MAC) address. This only works for for packets traversing the PREROUTING and INPUT chains. It provides only one option '`--mac-source`' as in

  ```
  iptables  -A INPUT  -m mac --mac-source 00:60:08:91:CC:B7  ACCEPT
  ```

  or as in

  ```
  iptables  -A INPUT  -m mac --mac-source ! 00:60:08:91:CC:B7  DROP
  ```

  The second rule will drop all incoming packets unless they are from the specific machine with the MAC address shown.

- Yet another useful extension module is the 'limit' module that is useful in warding of Denial of Service (DoS) attacks. This module is loaded into the kernel with the '`-m limit`' option. What the module does can be controlled by the subsequent option flags

'`--limit`' and '`--limit-burst`'. The following rule will limit a request for a new connection to one a second. Therefore, if DoS attack consists of bombarding your machine with SYN packets, this will get rid of most of them. This is referred to as "**SYN-flood protection**".

```
iptables -A FORWARD -p tcp --syn -m limit --limit 1/s -j ACCEPT
```

- The next rule is a protection against indiscriminate and nonstop scanning of the ports on your machine. This is referred to as protection against a "furtive port scanner":

```
iptables -A FORWARD -p tcp --tcp-flags SYN,ACK,FIN,RST \
                           -m limit --limit 1/s -j ACCEPT
```

- The next rule is a protection against what is called as the **"ping of death"** where someone tries to ping your machine in a non-stop fashion:

```
iptables -A FORWARD -p icmp --icmp-type echo-request \
                          -m limit --limit 1/s -j ACCEPT
```

Using `iptables` to do Port Forwarding

- Let's say that you have a firewall computer protecting a LAN. Let's also say that you are providing a web server on one of the LAN computers that is physically different from the firewall computer. Further, let's assume that there is a single IP address available for the whole LAN, this address being assigned to the firewall computer. Let's assume that this IP address is 123.45.67.89.

- So when a HTTP request comes in from the internet, it will typically be received on port 80 that is assigned to HTTP in `/etc/services`. The firewall would need to forward this request to the LAN machine that is actually hosting the web server.

- This is done by adding a rule to the PREROUTING chain of the NAT table:

  ```
  iptables -t nat  -A PREROUTING  -p tcp  -d 123.45.67.89 \
                -dport 80  -j DNAT  --to-destination 10.0.0.25
  ```

  where the jump target DNAT stands for **Dynamic Network Address Translation**. We are also assuming that the LAN address of the machine hosting the HTTP server is 10.0.0.25 in a Class A private network 10.0.0.0/8.

- If multiple LAN machines are simultaneously hosting the same HTTP server for reasons of high traffic to the server, you can spread the load of the service by providing a range of addresses for the '–to-destination' option, as by

```
      --to-destination 10.0.0.1-10.0.0.25
```

- This will now spread the load of the service over 25 machines, including the gateway machine if its LAN address is 10.0.0.1.

- So the basic idea in port forwarding is that you forward all the traffic received at a given port on our firewall computer to the designated machines in the LAN that is protected by the firewall.

Using Logging with `iptables`

- So far we have only talked about the following targets: ACCEPT, DENY, DROP, REJECT, REDIRECT, RETURN, and chain_name_to_jump_to for the **`filter`** table, and SNAT and DNAT for the **`nat`** table.

- One can also use LOG as a target. So if you did not want to drop a packet for some reason, you could go ahead and accept it but at the same time log it to decide later if your current rule for such packets is a good rule. Here is an example of a LOG target in a rule for the FORWARD chain:

```
iptables  -A FORWARD  -p tcp  -j LOG  --log-level info
```

- Here are all the possibilities for the '–log-level' argument:

```
emerg
alert
crit
err
warning
notice
info
```

```
        debug
```

- You can also supply a '–log-prefix' option to add further information to the front of all messages produced by the logging action:

```
    iptables  -A FORWARD  -p tcp  -j LOG  --log-level info  \
                                    --log-prefix "Forward INFO "
```

Experimenting with Firewalls on a Linux Machine

Let's say you are setting up a local area network and you designate a machine as a gateway between the internet and router that is connected to your LAN. You wish to experiment with different firewall rules in the gateway machine whose job is to protect the LAN from outside intrusions. We will refer to the gateway machine as the firewall computer. You execute the following steps in the firewall computer:

- You design a set of rules using the `iptables` based syntax shown earlier and place these rules in a shell executable file. The rest of the steps shown below are meant to make it possible for you to restore the original rules should something go wrong and to finally make the new rules as the operating rules. Assume that your new firewall rules are in a file called

  ```
  myfirewall.sh
  ```

  in any working directory.

- The firewall rules that the kernel is currently using are in the file `/etc/sysconfig/iptables`. Save these in a backup file by

  ```
  cd /etc/sysconfig
  ```

```
cp iptables iptables.original
```

Note that your system may use a shorthand notation for storing the rules in the file `/etc/sysconfig/iptables`. On my laptop, this file contains the following

```
# Firewall configuration written by system-config-securitylevel
# Manual customization of this file is not recommended.
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:RH-Firewall-1-INPUT - [0:0]
-A INPUT -j RH-Firewall-1-INPUT
-A FORWARD -j RH-Firewall-1-INPUT
-A RH-Firewall-1-INPUT -i lo -j ACCEPT
-A RH-Firewall-1-INPUT -p icmp --icmp-type any -j ACCEPT
-A RH-Firewall-1-INPUT -p 50 -j ACCEPT
-A RH-Firewall-1-INPUT -p 51 -j ACCEPT
-A RH-Firewall-1-INPUT -p udp --dport 5353 -d 224.0.0.251 -j ACCEPT
-A RH-Firewall-1-INPUT -p udp -m udp --dport 631 -j ACCEPT
-A RH-Firewall-1-INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
-A RH-Firewall-1-INPUT -j REJECT --reject-with icmp-host-prohibited
COMMIT
```

This format of the firewall rules is created by the command `iptables-save` when called on a **shell-executable file** that contains the actual invocations of the iptables commands for the individual rules.

- Stop the iptables service currently running on the firewall computer by

```
/etc/init.d/iptables stop
```

This will clear all existing firewall rules created previously at boot time.

- Now execute your shell file by

  ```
  sh myfirewall.sh
  ```

  This will load the new rules into the Linux kernel.

- To see how the rules were loaded into the Linux kernel, do the following

  ```
  iptables -L
  ```

  Since we did not specify the '-t' flag, by default it lists the contents of the filter table.

- If everything looks okay, save the rules that were loaded into the kernel by

  ```
  iptables-save  >  /etc/sysconfig/iptables
  ```

  From now on, each time you reboot the machine or restart iptables, the rules will be read from the file `/etc/sysconfig/iptables`

- Now restart iptables by

```
          /etc/init.d/iptables start
```

- If you are doing NAT, make sure you turn on IP packet forwarding by setting

```
        net.ipv4.ip_forward = 1
```

  in the `/etc/sysctl.conf` file.

- Some other points to remember:

  - Note that the names of built-in chains, INPUT, OUTPUT, and FORWARD, must always be in uppercase.

  - The '-p tcp' and '-p udp' options load into the kernel the TCP and UDP extension modules.

  - Chain names for user-defined chains can only be up to 31 characters.

  - User-defined chain names are by convention in lower-case.

  - When a packet matches a rule whose target is a user-defined chain, the packet begins traversing the rules in that user-defined chain. If that chain doesn't decide the fate of the packet, then once traversal on that chain has finished, traversal resumes on the next rule in the current chain.

– Even if the condition part of a rule is matched, if the rule does not specify a target, the next rule will be considered.

– User-defined chains can jump to other user-defined chains (but don't make loops: your packets will be dropped if they're found to be in a loop).

Some Simple Experiments with `iptables`: EXPERIMENT 1

- Let's say that my goal is to first turn off the ping on my Linux laptop (that is, I do not want the other machines to be able to ping my Linux laptop) and then to turn it back on. Note that a ping sends an ICMP type 8 (Echo Request) which all cooperative hosts should respond to with an ICMP type 0 (Echo Reply) packet.

- As you will recall, when my Linux laptop first boots up, it comes with the default RedHat firewall that was shown on Slide 11.

- Assuming the firewall shown on Slide 11 is up and running, would the following command

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

  block all incoming ping packets? **If you try it, you will notice that it does not.** You will still be able to ping the laptop from any other machine in the network. To see what is going on, let's execute the command

  ```
  iptables -L INPUT
  ```

  to see all of the rules in the **INPUT** chain. This returned

```
target               prot   opt  source   destination
-------------------------------------------------
RH-Firewall-1-INPUT  all    --  anywhere   anywhere
DROP                 icmp   --  anywhere   anywhere
```

The ordering of the rules implies that all incoming packets will be trapped by the first rule in the INPUT chain. So the incoming **icmp** packets will never see the second rule that was meant specifically for such packets. Of course, this happened because the '-A' option *appends* a new rule to the existing rules in a chain, that is, places it at the end of the chain.

- Let's therefore delete the newly inserted rule by using the same syntax as for rule insertion, except that we now replace the '-A' flag with the '-D' flag:

  ```
  iptables -D INPUT -p icmp -j DROP
  ```

  Now when we execute

  ```
  iptables -L INPUT
  ```

  we will NOT see the second rule in the table shown above. Note that we could also have deleted the rule by

  ```
  iptables -D INPUT 2
  ```

  where '2' stands for the second rule in the INPUT chain.

- So how do we insert the new rule for dropping the icmp packets so that it is at the top of the INPUT chain? To insert a rule at a specified position, we use the '-I' flag. So we can try the following next:

  ```
  iptables -I INPUT 1 -p icmp -j DROP
  ```

  This makes the new rule the first in the INPUT chain. This can be verified by executing again `iptables -L INPUT` to see all of the rules in the INPUT chain. This should now return

  ```
  target               prot   opt  source   destination
  -------------------------------------------------
  DROP                 icmp   --  anywhere   anywhere
  RH-Firewall-1-INPUT  all    --  anywhere   anywhere
  ```

  Now when you try to ping Linux laptop from any other machine, there will no response from the laptop.

- To restore the iptables, we would want to drop the new rule that is at position 1 by

  ```
  iptables -D INPUT 1
  ```

Some Simple Experiments with `iptables`: EXPERIMENT 2

- In this experiment, I want to be able to SSH into my Linux laptop. First, of course, I must make sure that you have the SSHD server daemon running on the laptop. That I can simply verify by

```
ps ax | grep sshd
```

- Making the same assumptions as those listed on Slide 13 and 14 with regard to the firewall that comes up when I boot up my Linux laptop, if I tried to SSH into the laptop (IP address: 192.168.1.128) by executing

```
ssh 192.168.1.128 -l kak
```

  on another machine in the same LAN, I'd get the following error message

```
ssh "no route to host"
```

- This error message was generated by rule #8 of the **RH-Firewall-1-INPUT** chain in the following output produced by the command '`iptables -L --line-numbers`' for the rules in the `filter` table:

```
Chain INPUT (policy ACCEPT)
num  target               prot opt source   destination
1    RH-Firewall-1-INPUT  all  --  anywhere anywhere

Chain FORWARD (policy ACCEPT)
num  target              prot  opt source   destination
1    RH-Firewall-1-INPUT  all  --  anywhere  anywhere

Chain OUTPUT (policy ACCEPT)
num  target              prot  opt source   destination

Chain RH-Firewall-1-INPUT (2 references)
num  target     prot opt source          destination
1    ACCEPT     all  --  anywhere        anywhere
2    ACCEPT     icmp --  anywhere        anywhere     icmp any
3    ACCEPT     ipv6-crypt --  anywhere  anywhere
4    ACCEPT     ipv6-auth --   anywhere  anywhere
5    ACCEPT     udp  --  anywhere        224.0.0.251  udp dpt:5353
6    ACCEPT     udp  --  anywhere        anywhere     udp dpt:ipp
7    ACCEPT     all  --  anywhere        anywhere     state RELATED,ESTABLISHED
8    REJECT     all  --  anywhere        anywhere     reject-with icmp-host-prohibited
```

- The request for the SSH connection from outside the laptop would get trapped by rule 8 of the user-defined chain **RH-Firewall-1-INPUT** with no return error message to the remote machine. In other words, the request for connection from the remote machine would simply disappear in the Linux laptop and the remote machine would think that there was no route to the Linux laptop. When you are first experimenting with firewalls, this may appear baffling since if you tried this since the following

  ```
  ping 192.168.1.128         (the address shown was the DHCP
                               address of the Linux laptop)
  ```

  always succeeded on the remotes machine. But looking at the

RH-Firewall-1-INPUT chain, it is clear as to why. All the icmp packets produced by ping were getting processed by line 2 of the **RH-Firewall-1-INPUT** chain.

- So I executed the following on the command line as root:

```
  iptables -I RH-Firewall-1-INPUT 8 -p tcp \
                     --destination-port 22 -j ACCEPT
```

  This rule, inserted at line 8 of the **RH-Firewall-1-INPUT** chain, says to accept all incoming TCP packets on the packets' destination port 22 (which as stated in `/etc/services` is the SSHD port). After the above change, what was originally rule 8 becomes rule 9. If we now call

```
      iptables -L --line-numbers
```

  again, I would see the following two for the lines 8 and 9 in the RH chain:

```
num  target   prot opt source      destination
-----------------------------------------------------
8    ACCEPT   tcp  --  anywhere    anywhere   tcp dpt:ssh
9    REJECT   all  --  anywhere    anywhere   reject-with icmp-host-prohibited
```

- After the above change, there should be no problem SSHing into the Linux laptop.

- Note also that rule 7 of the **RH-Firewall-1-INPUT** chain does not help us with the request for ssh connection because it only applies to already existing connections.

Some additional notes on getting SSHD to work through a firewall:

When I first started playing with getting SSHD to work on my laptop, every time I stopped and re-started the server daemon by

```
/etc/init.d/sshd stop
/etc/init.d/ssh start
```

the system would spit out an error message (that would typically be placed in `/var/log/secure`):

```
sshd error "bind to port 22 on 0.0.0.0 failed"
```

I believe I fixed this problem by making the following two changes to the `/etc/ssh/sshd_config` file: I made sure that the following line was uncommented

```
ListenAddress 0.0.0.0
```

and that the following line was commented out as shown below

```
#ListenAddress ::
```

This line is for IPv6 addresses. If both the above lines in `/etc/ssh/sshd_config` are commented out, as is the case with the original version of this configuration file, by default the system tries to bind the server daemon a second time on port 22 for IPv6 after the binding has already succeeded for IPv4.

Additionally, when you are debugging in the mode shown above, it is also a good idea to check `/var/log/messages` for error messages. You may also wish to make an SSH connection in the verbose mode by, say,

```
ssh -v 192.168.1.128 -l kak
```

for a verbose report on the connection process. For an even more verbose report, try

```
ssh -vvv 192.168.1.128 -l kak
```

Case Study: Designing Iptables for a New LAN

Let's say that you want to create a firewall to protect a Class C 192.168.1.0/24 private LAN that is connected to the rest of the internet by a router and a gateway machine as shown.

```
  local_machine_1   local_machine_2   .....   local_machine_N

          |               |                          |    LAN addresses:
          \               |                          /     192.168.1.0/24
           \              |                         /
            \             |                        /
              -----------------------------------
             |              ROUTER               |
              -----------------------------------
                              |
                              |
                              |
                              |  interface eth1, IP adress: 192.168.1.1
                   -----------------------
                  |                       |
                  |     Gateway           |  loopback: localhost
                  |     Machine           |  addess:   127.0.0.1
                  |  (firewall computer)  |  interface: lo
                  |                       |
                   -----------------------
                              |  interface eth0, IP address: 123.45.67.89
                              |
                              |
                          internet
```

We will also assume that the gateway machine has its IP address assigned dynamically (DHCP) by some ISP. We will assume that the gateway machine is using Linux as its OS and that **iptables** based packet filtering software is installed. We want the firewall installed in the gateway machine to allow for the following:

- It should allow for unrestricted internet access from all the machines in the LAN.
- Allow for SSH access (port 22) to the firewall machine from outside the LAN for external maintenance of this machine.
- Permit `Auth/Ident` (port 113) that is used by some services like SMTP and IRC. (Note that port 113 is for Auth (authentication). Some old servers try to identify a client by connecting back to the client machine on this port and waiting for the IDENTD server on the client machine to report back. But this port is now considered to be a security hole. See http://www.grc.com/port_113.htm. So, for this port, 'ACCEPT' should probably be changed to DROP.)
- Let's say that the LAN is hosting a web server (on behalf of the whole LAN) and that this HTTPD server is running on the machine 192.168.1.100 of the LAN. So the firewall must use NAT to redirect the incoming TCP port 80 requests to 192.168.1.100.

- We also want the firewall to accept the ICMP Echo requests (as used by ping) coming from the outside.

- The firewall must log the filter statistics on the external interface of the firewall machine.

- We want the firewall to respond back with TCP RST or ICMP Unreachable for incoming requests for blocked ports.

- Shown below is Rusty Russell's recommended firewall that has the above mentioned features:

```
#! /bin/sh

# macro for external interface:
ext_if = "eth0"
# macro for internal interface:
int_if = "ath0"

tcp_services = "22,113"
icmp_types = "ping"

comp_httpd = "192.168.1.100"

# NAT/Redirect
modprobe ip_nat_ftp
iptables -t nat -A POSTROUTING -o $ext_if -j MASQUERADE
iptables -t nat -i -A PREROUTING $ext_if -p tcp --dport 80 -j DNAT --to-destination $comp

# filter table rules
# Forward only from external to webserver:
iptables -A FORWARD -m state --state=ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -i $ext_if -p tcp -d $comp_httpd --dport 80 --syn -j ACCEPT
```

```
# From internal is fine, rest rejected
iptables -A FORWARD -i $int_if -j ACCEPT
iptables -A FORWARD -j REJECT

# External can only come in to $tcp_services and $icmp_types
iptables -A INPUT -m state --state=ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -i $ext_if -p tcp --dport $tcp_services --syn -j ACCEPT
for icmp in $icmp_types; do
    iptables -A INPUT -p icmp --icmp-type $icmp -j ACCEPT
done

# Internal and loopback are allowed to send anything:
iptables -A INPUT -i $int_if -j ACCEPT
iptables -A INPUT -i lo -j ACCEPT
iptables -A INPUT -j REJECT

# logging
echo "1" > /proc/sys/net/ipv4/ip_forward
```