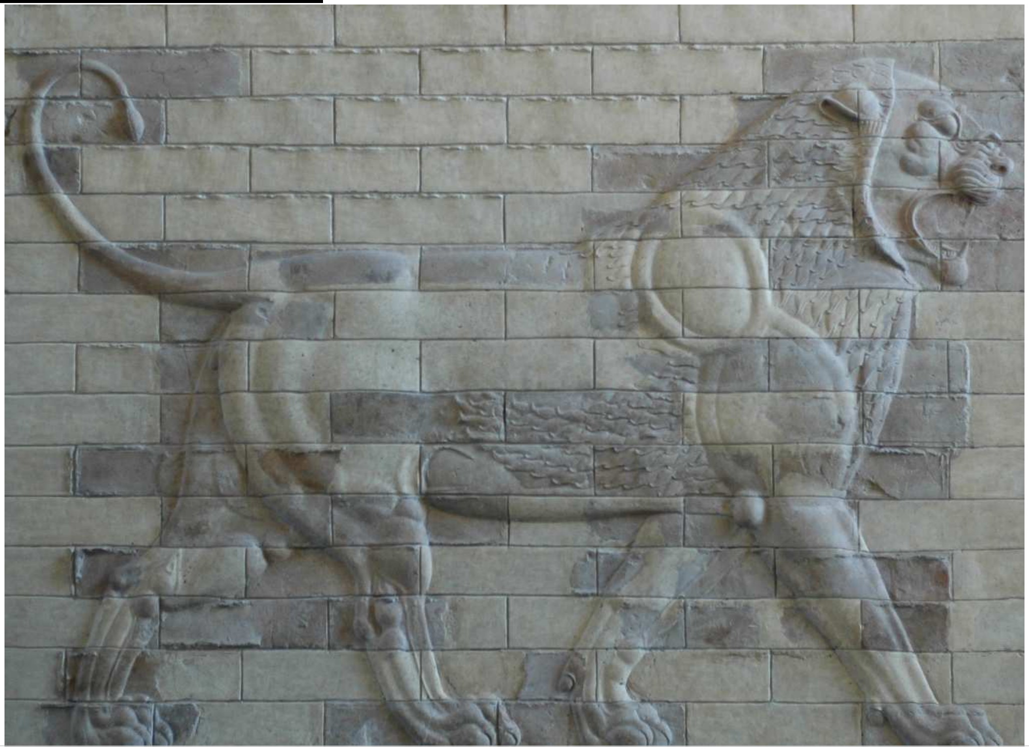
Linux for Researchers

Chapter 8: Firewalls and Services

Now that we've discussed networking in some detail, it's time to talk about network services. Along with that comes concern about security, so we'll also start looking at firewall configurations.

This will be hodgepodge of topics, but they all revolve around those two themes: servers and firewalls.

Part 1: Firewalls



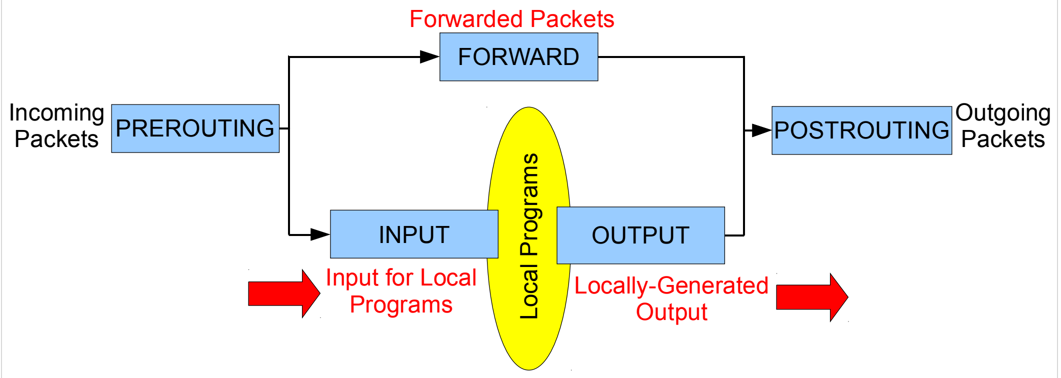
In the terminology we'll use today, a firewall is anything that blocks or modifies network traffic. Most desktop computers today have some sort of firewall capability. They can, for example, selectively block incoming IP packets.

Even if your computer is behind a department firewall, or is running other security software, it's very important to have a properly-configured local firewall on your computer. This reflects a security philosophy called “defense-in-depth”, which says that you need multiple layers of defense. Multiple layers provide redundancy, in case one layer fails, and they tend to fill in the gaps in each other's coverage.

Netfilter:

Built into the Linux kernel is a system called "Netfilter" that allows for monitoring, modifying or blocking IP packets as they pass through the kernel, based on packet header information. Netfilter associates user-defined functions with pre-defined "hooks" at various points along a packet's path through the kernel. These functions are managed by programs like "iptables".

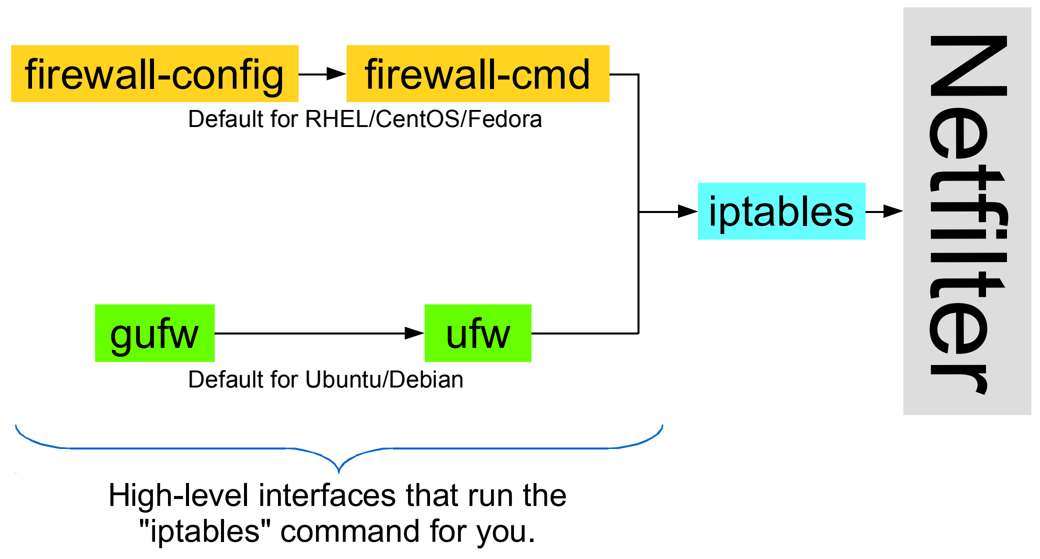
The diagram below shows some of the available hooks, in blue:



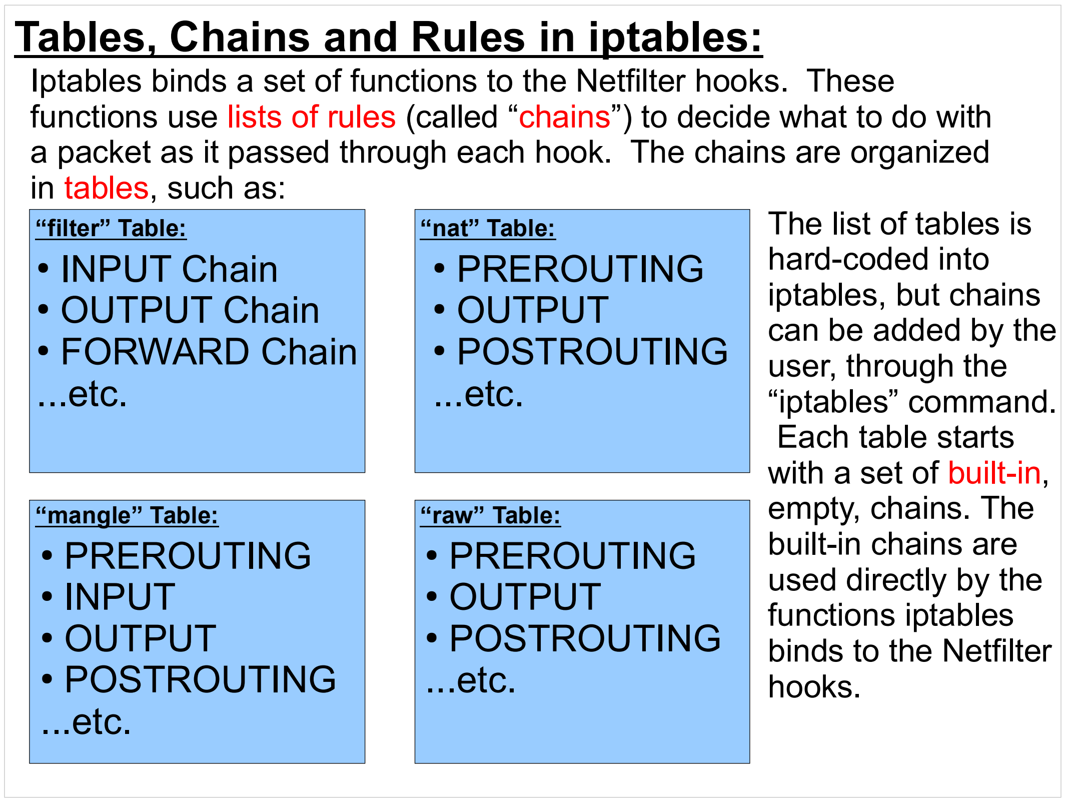
Netfilter is just a framework within the kernel. To use it, you need a program like iptables.

The input and output hooks let you filter traffic going to or coming out of local program. The forward hook allows you to filter network traffic that's just passing through your computer. The prerouting and postrouting hooks allow you to do things like re-writing the address on incoming/outgoing packets.

Tools for Controlling Netfilter:


firewall-config
firewall-cmd
Default for RHEL/CentOS/Fedora
gufw
ufw
Default for Ubuntu/Debian
iptables
Netfilter
High-level interfaces that run the "iptables" command for you.

Most Linux distributions provide tools for manipulating firewall rules at a very abstract level, but these tools generally are just wrappers that use the iptables command to actually do the work. Today we'll be using iptables directly, to give us a better understanding of how firewalls really work.



Tables, Chains and Rules in iptables:

Iptables binds a set of functions to the Netfilter hooks. These functions use lists of rules (called "chains") to decide what to do with a packet as it passed through each hook. The chains are organized in tables, such as:

"filter" Table:
- INPUT Chain
- OUTPUT Chain
- FORWARD Chain

...etc.

"nat" Table:
- PREROUTING
- OUTPUT
- POSTROUTING

...etc.

"mangle" Table:
- PREROUTING
- INPUT
- OUTPUT
- POSTROUTING

...etc.

"raw" Table:
- PREROUTING
- OUTPUT
- POSTROUTING

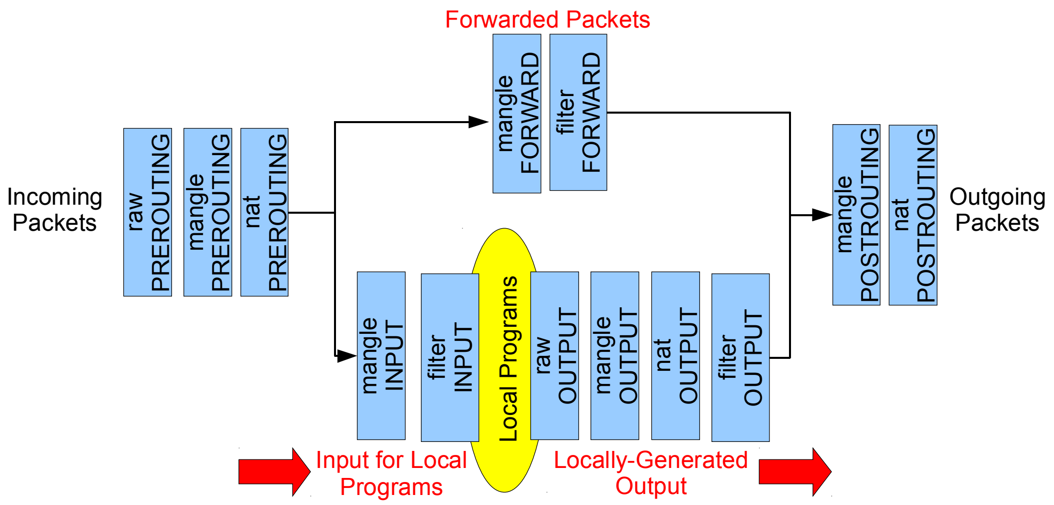
...etc.

The list of tables is hard-coded into iptables, but chains can be added by the user, through the "iptables" command. Each table starts with a set of built-in, empty, chains. The built-in chains are used directly by the functions iptables binds to the Netfilter hooks.

Note that the names of tables and chains are case-sensitive.

How iptables Chains are Bound to Netfilter Hooks:

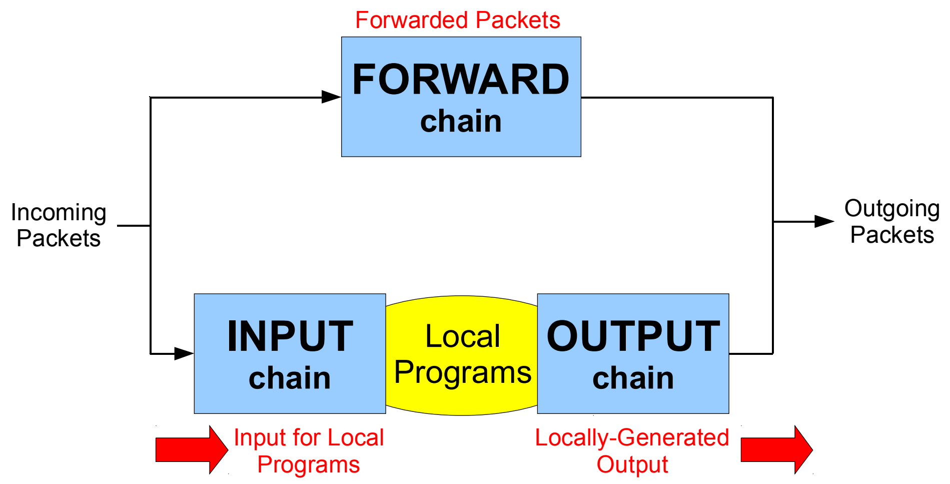
Here's how iptables binds various built-in chains to the Netfilter hooks:



This shows where the iptables chains from the previous slide plug into the hooks provided by Netfilter.

The “filter” Table:

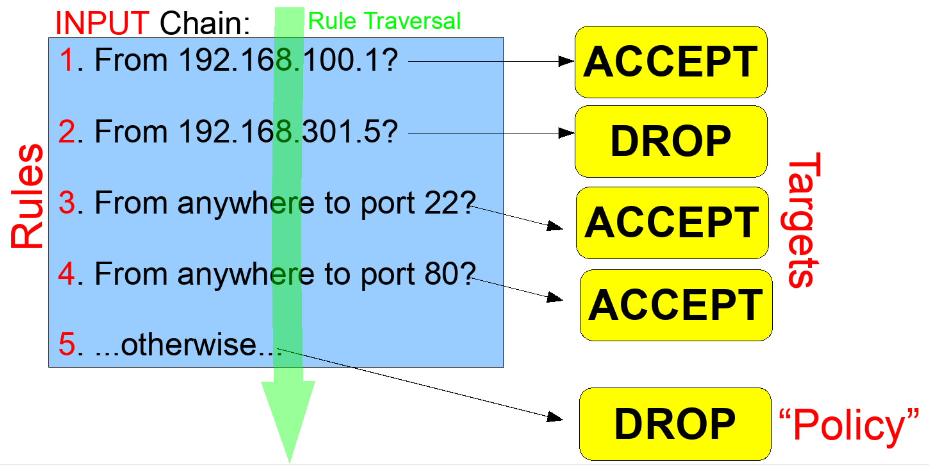
The most often-used table is the “filter” table, which initially contains built-in chains called “INPUT”, “OUTPUT” and “FORWARD”. These chains of rules are used by functions plugged into the Netfilter hooks shown below:



Remember, again, that these names are case-sensitive. It's “INPUT”, not “input”.

Iptables Chains:

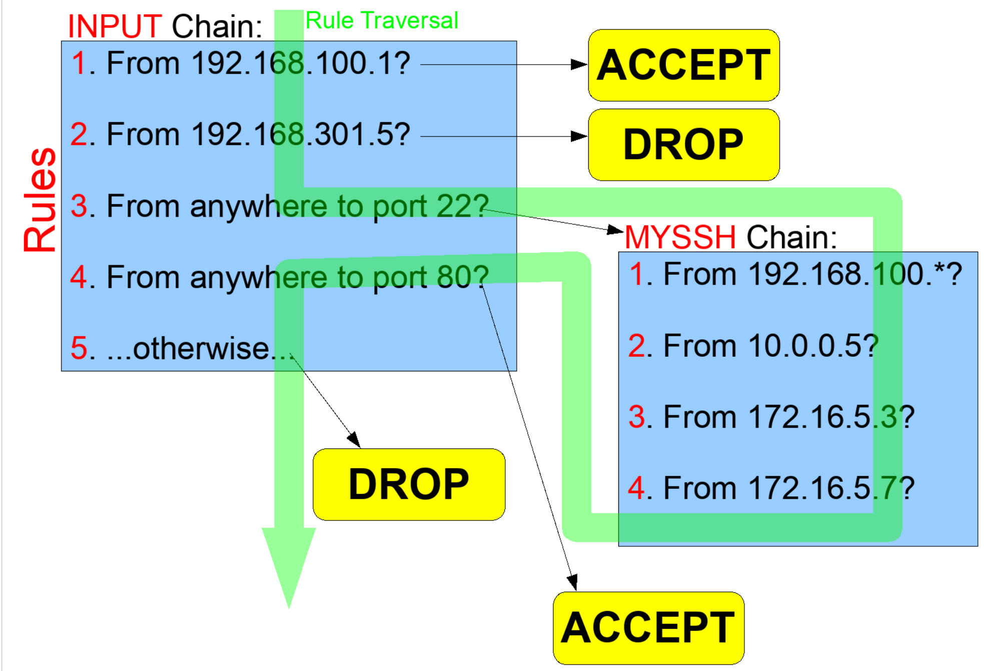
Each iptables chain is a list of rules. Each rule consists of a test and a "target". The target can either be selected from a list of built-in targets (e.g., "ACCEPT" or "DROP"), or it can point to a different, user-defined, chain of other rules. The rules in the chain are processed ("traversed") from top to bottom, like a program. Some targets (like "DROP") will cause the "program" to halt. Others (like "LOG") will allow the "program" to keep running. Built-in chains each have a "policy" that determines what happens to packets that reach the end of the chain.



Targets that cause rule traversal to stop are called "terminating" targets. Those that don't are called "non-terminating" targets.

Only built-in chains have policies. The built-in chains are the ones that are directly attached to Netfilter's hooks. User-defined chains are always called by one of the built-in chains.

User-Defined Chains as Targets:



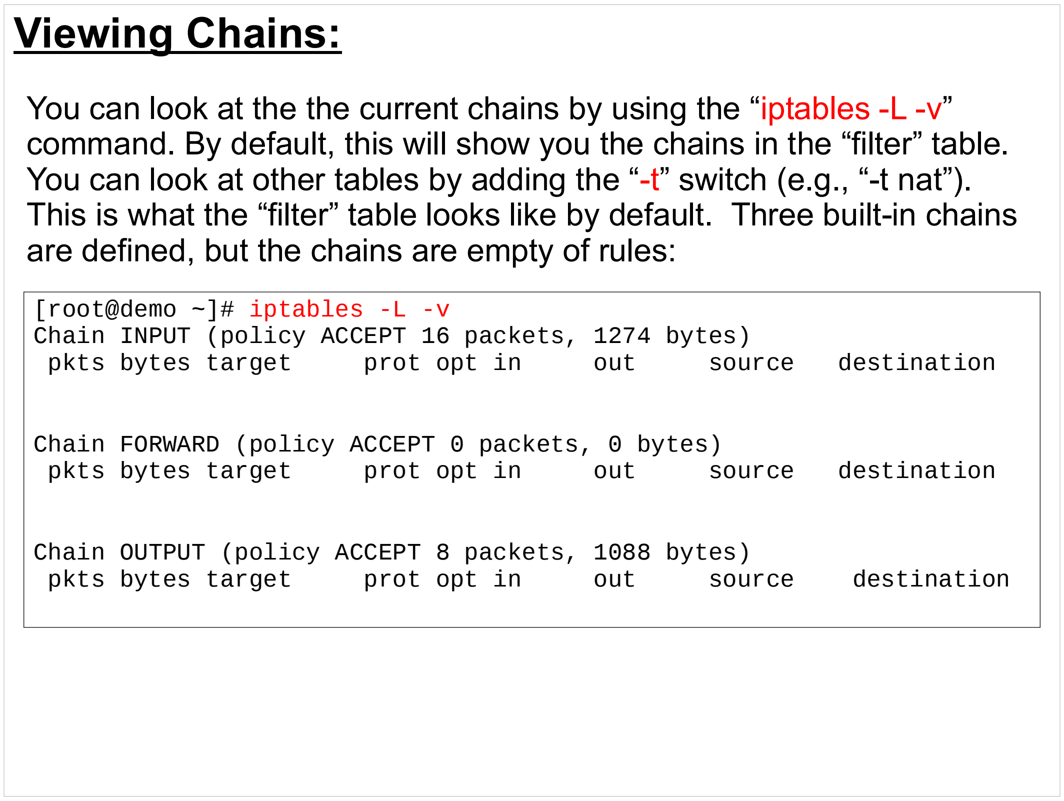
Here, the INPUT chain is a built-in chain, and the MYSSH chain is user-defined. To save space, I've omitted the targets for the MYSSH chain, but these rules would actually have targets also.

Some iptables Targets:

Here are some examples of built-in iptables targets:

ACCEPT	Stop traversal, allow the packet to continue.
DROP	Stop traversal, ignore the packet.
REJECT	Stop traversal, ignore the packet, but notify the sender.
LOG	Log the packet, then continue traversal.
TARPIT	Wait forever without responding to sender (TCP only).

...etc.



Viewing Chains:

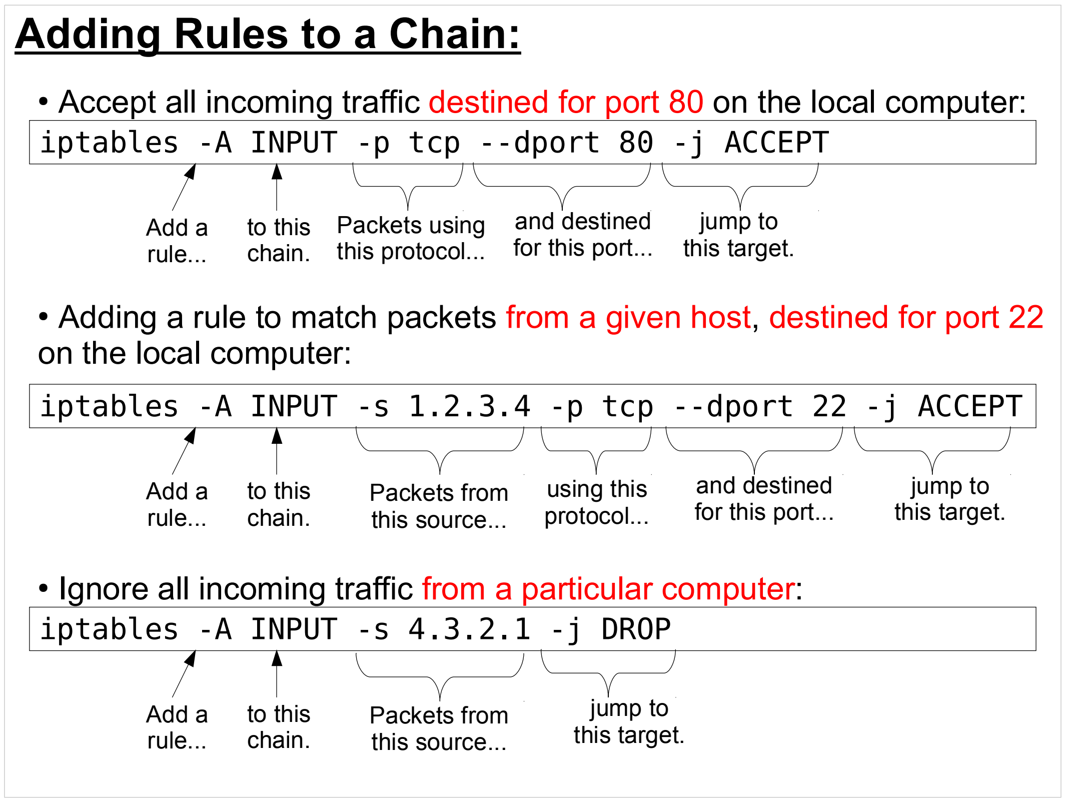
You can look at the the current chains by using the “iptables -L -v” command. By default, this will show you the chains in the “filter” table. You can look at other tables by adding the “-t” switch (e.g., “-t nat”). This is what the “filter” table looks like by default. Three built-in chains are defined, but the chains are empty of rules:

```
[root@demo ~]# iptables -L -v
Chain INPUT (policy ACCEPT 16 packets, 1274 bytes)
 pkts bytes target     prot opt in     out     source    destination


Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source    destination


Chain OUTPUT (policy ACCEPT 8 packets, 1088 bytes)
 pkts bytes target     prot opt in     out     source     destination
```

You can also just use “iptables -L”, but if you have non-trivial firewall rules you'll find that the output is misleading. For one thing, “iptables -L” doesn't tell you which network interfaces a given rule applies to.



Adding Rules to a Chain:

- Accept all incoming traffic destined for port 80 on the local computer:

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

Add a rule...
to this chain.
Packets using this protocol...
and destined for this port...
jump to this target.

- Adding a rule to match packets from a given host, destined for port 22 on the local computer:

```
iptables -A INPUT -s 1.2.3.4 -p tcp --dport 22 -j ACCEPT
```

Add a rule...
to this chain.
Packets from this source...
using this protocol...
and destined for this port...
jump to this target.

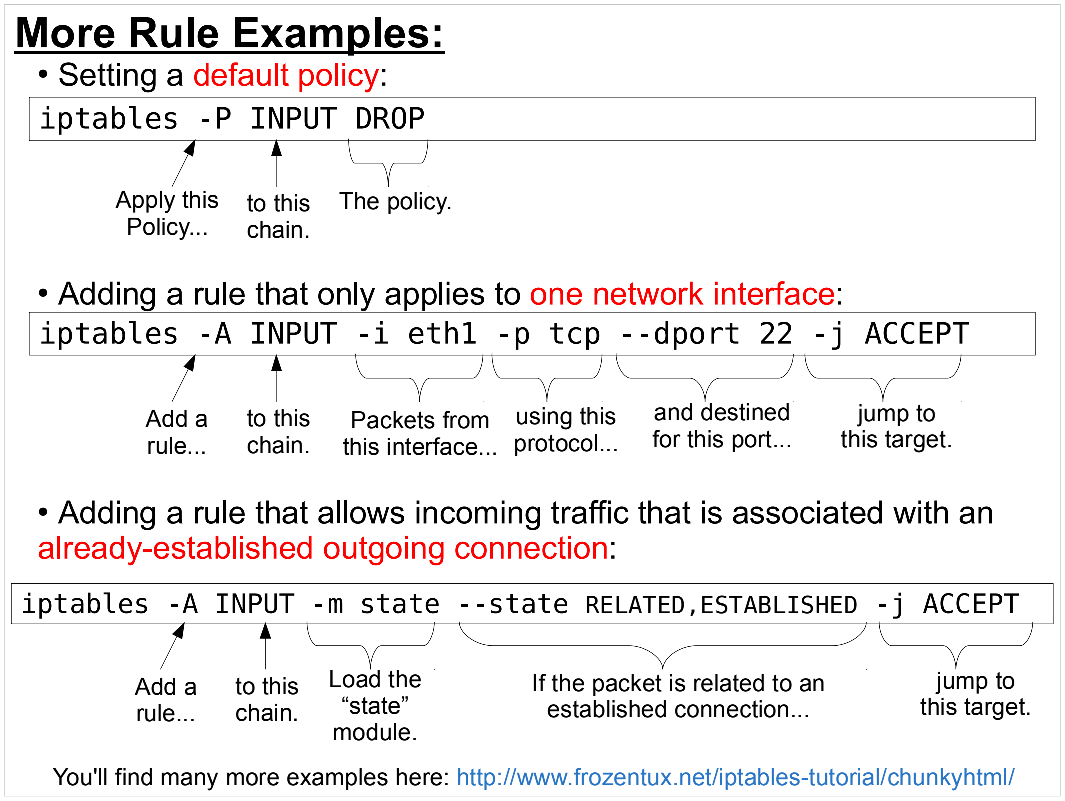
- Ignore all incoming traffic from a particular computer:

```
iptables -A INPUT -s 4.3.2.1 -j DROP
```

Add a rule...
to this chain.
Packets from this source...
jump to this target.

A few simple examples. Later, we'll see how to define firewall rules automatically at boot time.

Note that "-A" appends the rule to the end of the chain. If you want to insert a rule at the top of the chain, use "`-I`" instead.



More Rule Examples:

• Setting a default policy:

```
iptables -P INPUT DROP
```

• Adding a rule that only applies to one network interface:

```
iptables -A INPUT -i eth1 -p tcp --dport 22 -j ACCEPT
```

• Adding a rule that allows incoming traffic that is associated with an already-established outgoing connection:

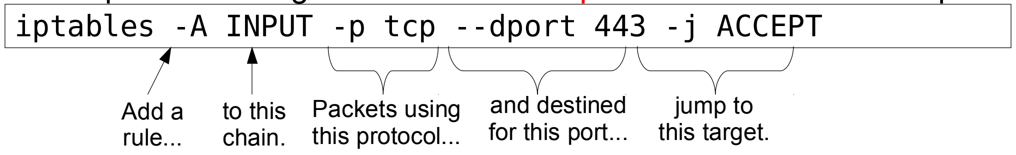
```
iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
```

You'll find many more examples here: http://www.frozentux.net/iptables-tutorial/chunkyhtml/

As you can see from the last example, iptables can be extended through “modules”. Many of these modules are already installed in most Linux distributions. These make iptables very powerful. You can, for example, do rate limiting, or limit the number of connections from a given host. You can filter by MAC address. You can select every nth packet (!). You can assign tags to packets for use in later rules. You can filter packets based on their length. You can even match strings within packets.

More Rule Examples:

• Accept all incoming traffic destined for port 443 on the local computer:

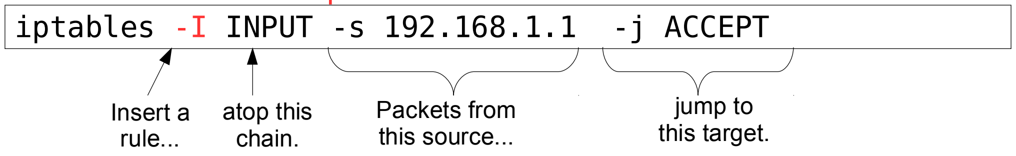


```
iptables -A INPUT -p tcp --dport 443 -j ACCEPT
```

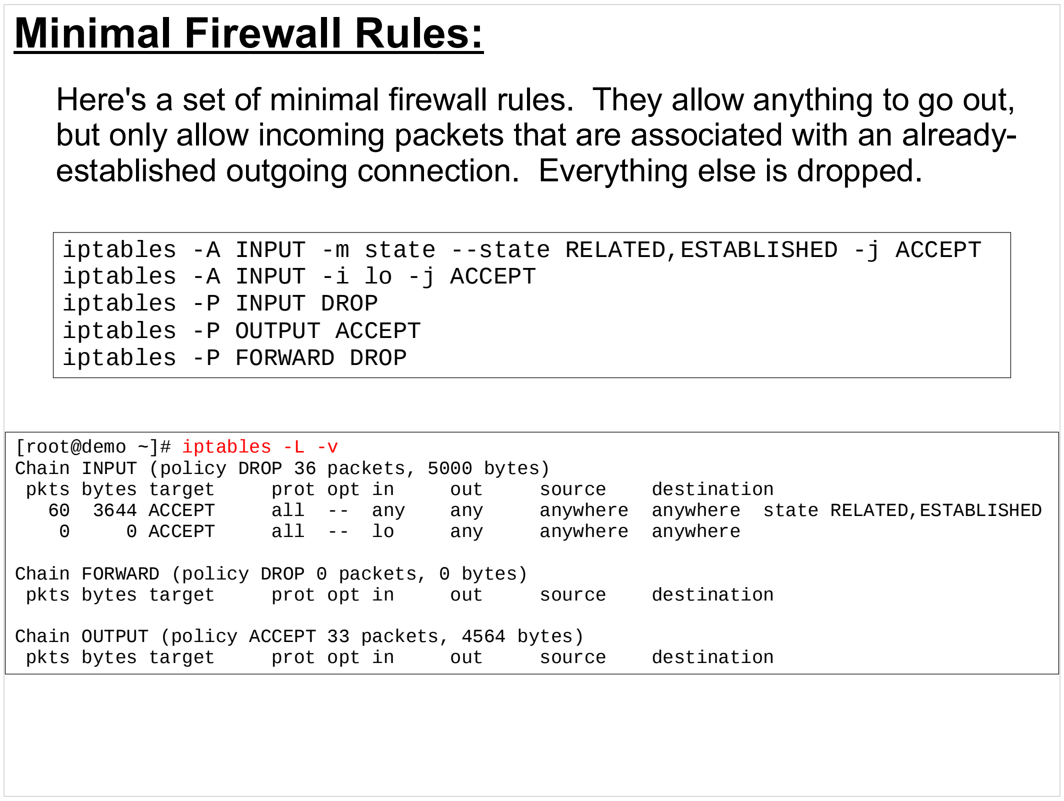
• Delete the rule created above:

```
iptables -D INPUT -p tcp --dport 443 -j ACCEPT
```

• Insert a rule at the top of the chain:



```
iptables -I INPUT -s 192.168.1.1  -j ACCEPT
```



Minimal Firewall Rules:

Here's a set of minimal firewall rules. They allow anything to go out, but only allow incoming packets that are associated with an already-established outgoing connection. Everything else is dropped.

```
iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables -A INPUT -i lo -j ACCEPT
iptables -P INPUT DROP
iptables -P OUTPUT ACCEPT
iptables -P FORWARD DROP
```

```
[root@demo ~]# iptables -L -v
Chain INPUT (policy DROP 36 packets, 5000 bytes)
 pkts bytes target     prot opt in     out     source    destination
   60  3644 ACCEPT     all  --  any    any     anywhere  anywhere   state RELATED,ESTABLISHED
    0     0 ACCEPT     all  --  lo     any     anywhere  anywhere

Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source    destination

Chain OUTPUT (policy ACCEPT 33 packets, 4564 bytes)
 pkts bytes target     prot opt in     out     source    destination
```

This is similar to the default firewall rules you'll find under Red Hat/Fedora/CentOS, or in any home internet router/firewall.

The “iptables-save” and “iptables-restore” Tools:

The firewall rules you create with iptables are volatile. They won't automatically be restored the next time you restart your computer, unless you take steps to restore them. One mechanism for doing this is the “iptables-save” and “iptables-restore” commands. If you've configured a set of firewall rules and want to save that configuration, issue a command like:

```
[root@demo ~]# iptables-save > myfirewall.conf
```

Then you can restore these rules later by typing:

```
[root@demo ~]# iptables-restore < myfirewall.conf
```

The output of iptables-save is just text, and can be edited with any text editor. It looks like this:

```
# Generated by iptables-save v1.3.5 on Tue Mar  3 14:38:46
2009
*filter
:INPUT DROP [18:2119]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [28:2832]
-A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
-A INPUT -i lo -j ACCEPT
COMMIT
# Completed on Tue Mar  3 14:38:46 2009
```

Iptables Configuration Files:

To save your iptables configuration, either use one of the high-level configuration utilities like ufw or firewall-config, or...

On Red Hat/Fedora/CentOS firewall rules can be stored in:

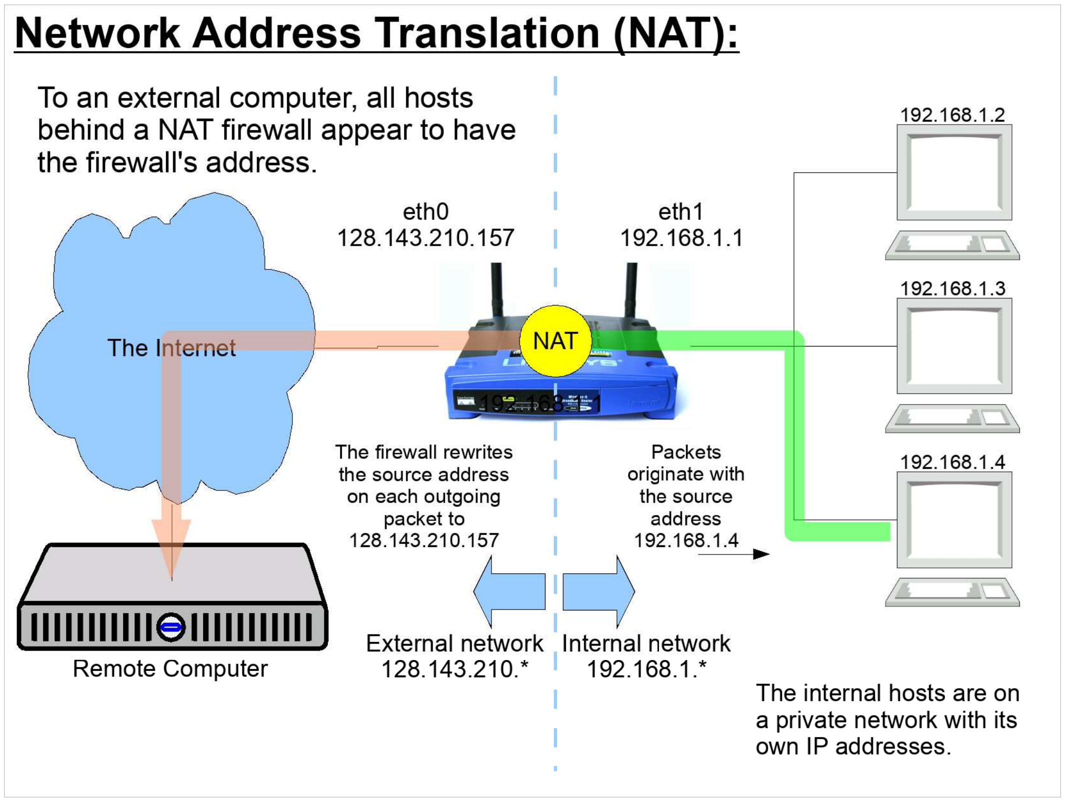
/etc/sysconfig/iptables

At boot time, this file is automatically read by iptables-restore to set up firewall rules if the "iptables" service is enabled.

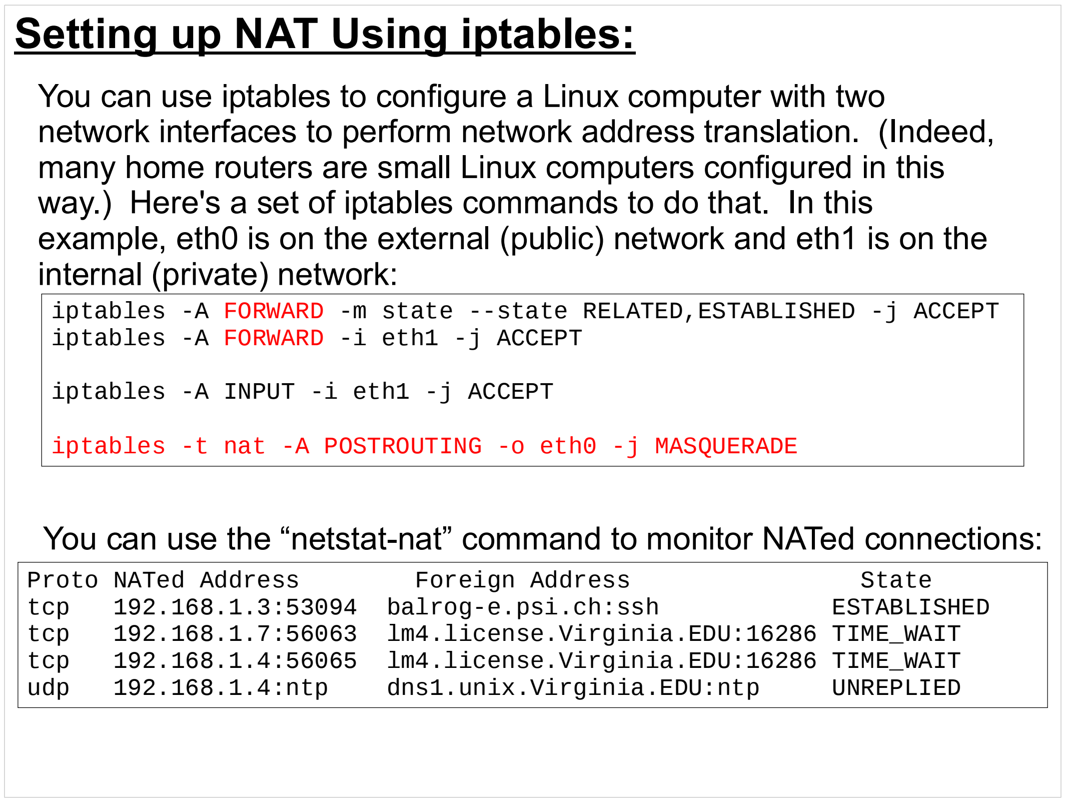
Under Ubuntu you can install the package iptables-persistent, which will automatically read firewall rules from:

/etc/iptables/rules/rules.v4

Network Address Translation (NAT):


To an external computer, all hosts behind a NAT firewall appear to have the firewall's address.
eth0
128.143.210.157
eth1
192.168.1.1
192.168.1.2
192.168.1.3
192.168.1.4
The Internet
NAT
The firewall rewrites the source address on each outgoing packet to 128.143.210.157
Packets originate with the source address 192.168.1.4
Remote Computer
External network 128.143.210.*
Internal network 192.168.1.*
The internal hosts are on a private network with its own IP addresses.

Inexpensive home routers use NAT to connect computers in your home to the Internet. Many of these routers are actually running Linux, and use iptables, just as you'd use it on your desktop computer or a Linux server.



Setting up NAT Using iptables:

You can use iptables to configure a Linux computer with two network interfaces to perform network address translation. (Indeed, many home routers are small Linux computers configured in this way.) Here's a set of iptables commands to do that. In this example, eth0 is on the external (public) network and eth1 is on the internal (private) network:

```
iptables -A FORWARD -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables -A FORWARD -i eth1 -j ACCEPT

iptables -A INPUT -i eth1 -j ACCEPT

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

You can use the “netstat-nat” command to monitor NATed connections:

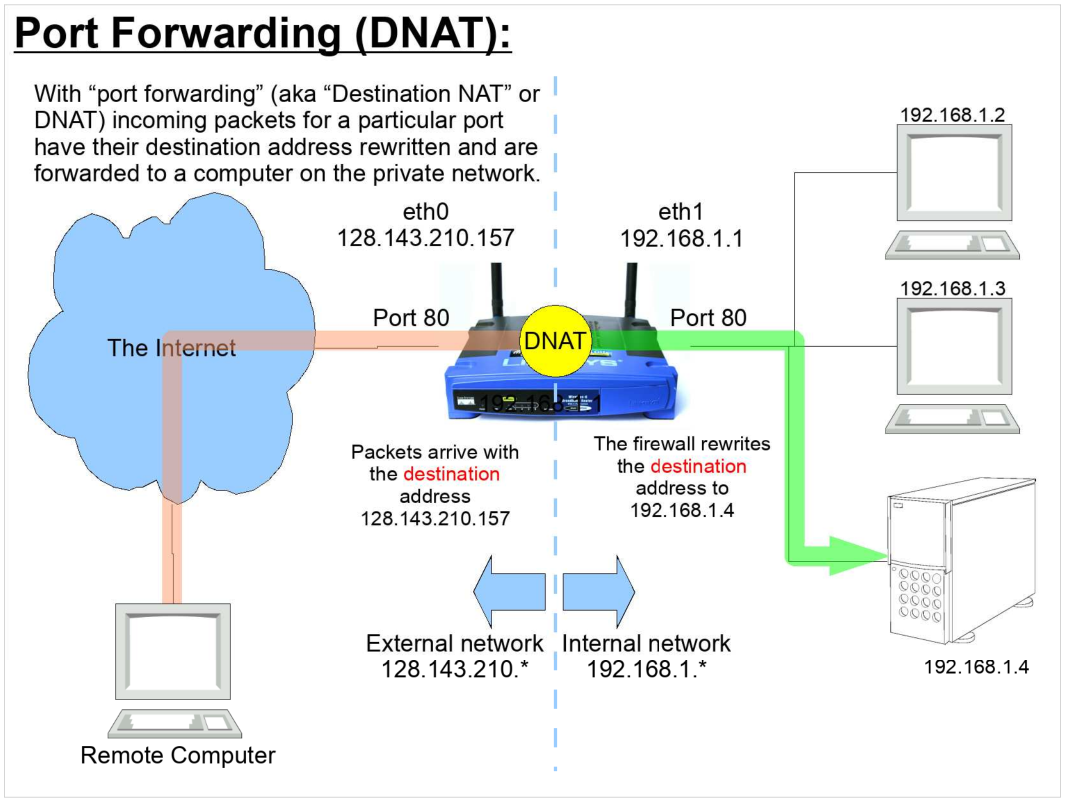
```
Proto NATed Address        Foreign Address                State
tcp   192.168.1.3:53094   balrog-e.psi.ch:ssh            ESTABLISHED
tcp   192.168.1.7:56063   lm4.license.Virginia.EDU:16286 TIME_WAIT
tcp   192.168.1.4:56065   lm4.license.Virginia.EDU:16286 TIME_WAIT
udp   192.168.1.4:ntp     dns1.unix.Virginia.EDU:ntp     UNREPLIED
```

The netstat-nat command is similar to the netstat command we looked at earlier, except that it shows you information about NATed connections passing through your computer.

This type of NAT is also called “source NAT”, or SNAT, since it re-writes the address of the source computer. As we'll see, there's also “destination NAT” or DNAT.

iptables actually has two possible targets for source NAT. The one shown above, MASQUERADE, is appropriate for devices that have variable IP addresses, supplied by a DHCP server. The other target is SNAT, which is more appropriate for hosts with fixed IP addresses. See the iptables man page for more information about the differences between the two.

Port Forwarding (DNAT):


With "port forwarding" (aka "Destination NAT" or DNAT) incoming packets for a particular port have their destination address rewritten and are forwarded to a computer on the private network.
eth0
128.143.210.157
eth1
192.168.1.1
Port 80
Port 80
DNAT
The Internet
192.168.1.2
192.168.1.3
192.168.1.4
Packets arrive with the destination address 128.143.210.157
The firewall rewrites the destination address to 192.168.1.4
External network
128.143.210.*
Internal network
192.168.1.*
Remote Computer

You could use port forwarding to connect a home web server to the Internet, for example. The details of how to do this will depend on the particular network hardware you have at home. In general, you'll need to connect to your router or DSL modem (or both) through these devices' web interfaces and configure NAT appropriately. If you have a DSL modem and a router, you may need to tell the DSL modem to forward packets to the router, and then tell the router to forward packets to your internal server. Documentation for most of these devices can be found on the web.

Setting up Port Forwarding Using iptables:

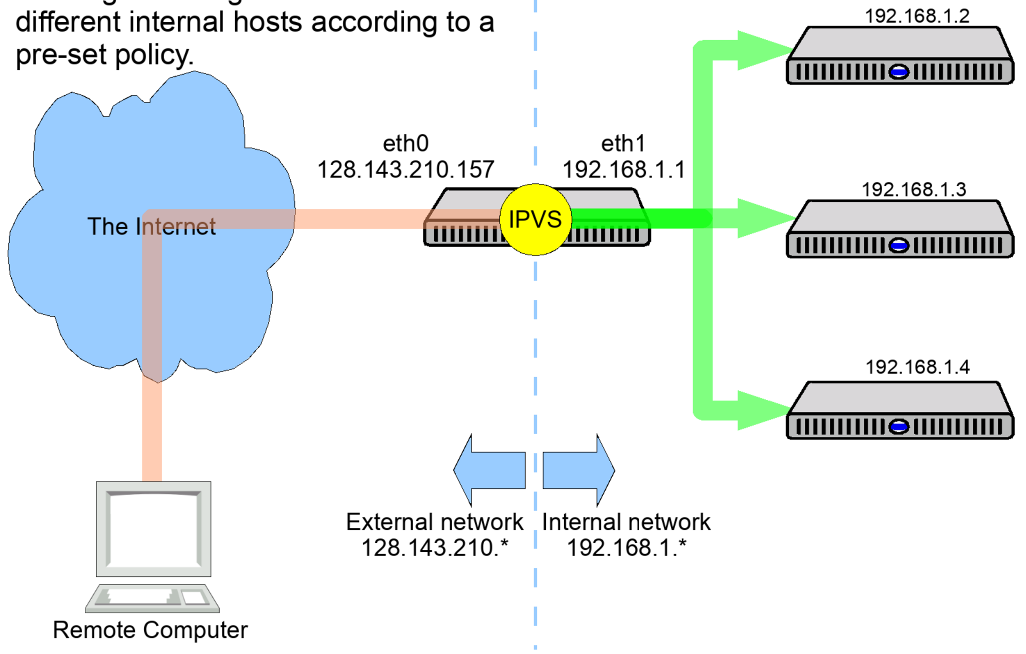
Port forwarding can also be done with the rules in the “nat” table. Again, eth0 is on the external (public) network. The host 192.168.1.4 is a web server on the internal (private) network. The rule below forwards incoming traffic bound for port 80 (the standard port for web traffic) to the internal host.

```
iptables -t nat -A PREROUTING -i eth0 -p tcp \
--dport 80 -j DNAT --to-destination 192.168.1.4
```

Here we see an iptables target (DNAT) that requires an argument. In this case, we need to specify the address of the internal computer to which we want to send the packets.

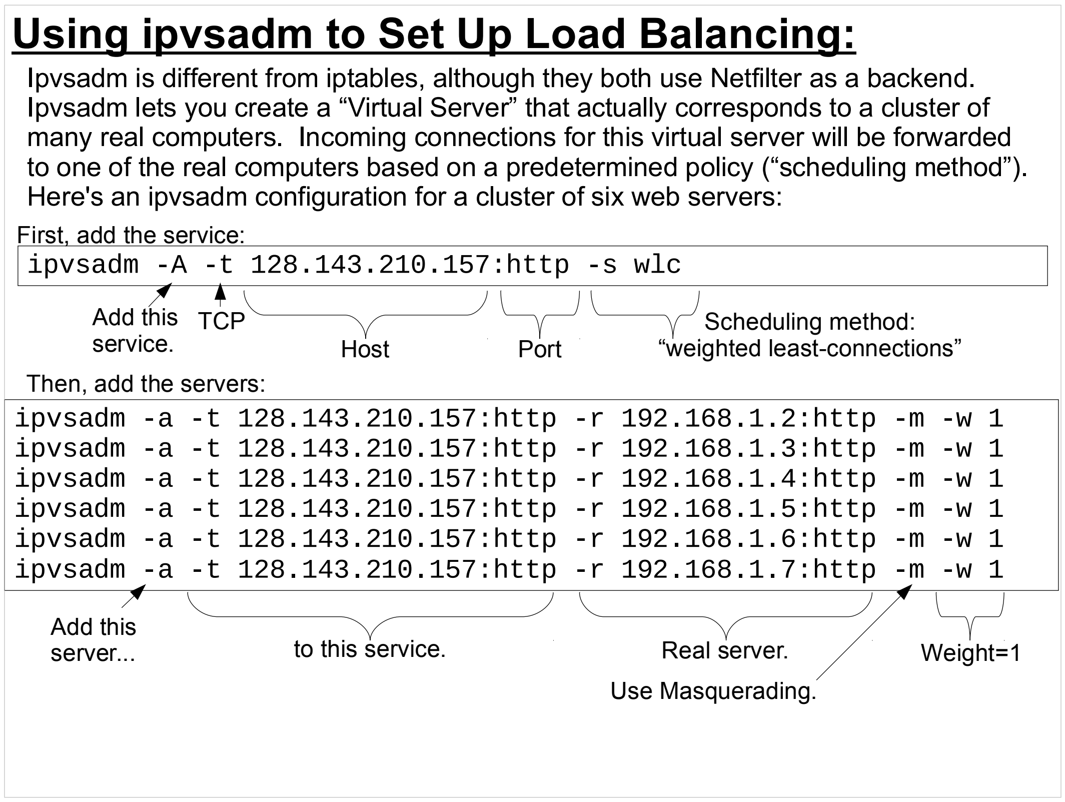
Load Balancing with “ipvsadm”:

ipvs does dynamic port forwarding, sending incoming connections to different internal hosts according to a pre-set policy.



If you were running an Internet business and you expected a lot of traffic on your web servers, you might want to be able to spread the traffic around, so that the load is handled by several web servers. IPVS is one way of doing this.

IPVS doesn't use iptables, but they both use the underlying Netfilter framework.



Using ipvsadm to Set Up Load Balancing:

Ipvsadm is different from iptables, although they both use Netfilter as a backend. Ipvsadm lets you create a “Virtual Server” that actually corresponds to a cluster of many real computers. Incoming connections for this virtual server will be forwarded to one of the real computers based on a predetermined policy (“scheduling method”). Here's an ipvsadm configuration for a cluster of six web servers:

First, add the service:

```
ipvsadm -A -t 128.143.210.157:http -s wlc
```

Then, add the servers:

```
ipvsadm -a -t 128.143.210.157:http -r 192.168.1.2:http -m -w 1
ipvsadm -a -t 128.143.210.157:http -r 192.168.1.3:http -m -w 1
ipvsadm -a -t 128.143.210.157:http -r 192.168.1.4:http -m -w 1
ipvsadm -a -t 128.143.210.157:http -r 192.168.1.5:http -m -w 1
ipvsadm -a -t 128.143.210.157:http -r 192.168.1.6:http -m -w 1
ipvsadm -a -t 128.143.210.157:http -r 192.168.1.7:http -m -w 1
```

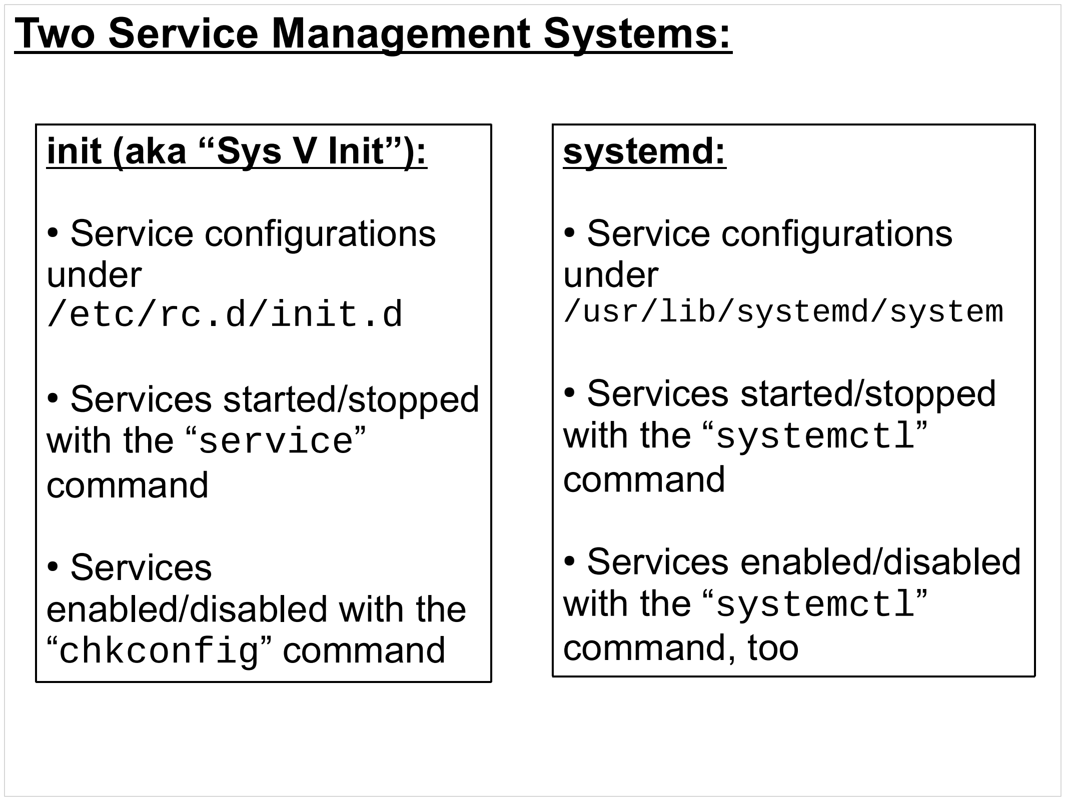
Like iptables, there are ipvsadm-save and ipvsadm-restore commands to save and restore an ipvsadm configuration. In the Red Hat/Fedora/CentOS world, the file /etc/sysconfig/ipvsadm will automatically be used to configure ipvsadm at boot time if the ipvsadm service is turned on.

Available scheduling methods include round-robin, fixed target based on source address, and many others in addition to the wlc method shown above.

Part 2: Services

TON GLOBE—SATURDAY, MAY 28, 1898.
BILL OF FARE
CUBA STEAK
PORTO RICO PIG
PHILIPPINE FLOATING ISLANDS
SANDWICH ISLANDS
McKINLEY
WELL, I HARDLY KNOW WHICH TO TAKE FIRST!

Now we'll look at how services are started and stopped, including how to start them automatically when the computer is booted.



Two Service Management Systems:

init (aka “Sys V Init”):

• Service configurations under `/etc/rc.d/init.d`

• Services started/stopped with the “`service`” command

• Services enabled/disabled with the “`chkconfig`” command

systemd:

• Service configurations under `/usr/lib/systemd/system`

• Services started/stopped with the “`systemctl`” command

• Services enabled/disabled with the “`systemctl`” command, too

Most current Linux distributions use systemd, but older systems used init. In many cases, systemd can understand "legacy" init-style service configuration files, letting you start/stop those services with systemd's own tools, just as if they were native systemd services.

Starting and Stopping Services with Systemd:

This service is named “ssh” on current Ubuntu/Debian systems.

Start this service now:

```
systemctl start sshd
```

Stop this service now:

```
systemctl stop sshd
```

Configuring Services to Start Automatically:

Automatically start this service at boot time:

```
systemctl enable sshd
```

Do not automatically start this service at boot time:

```
systemctl disable sshd
```

Checking Status of a Service with Systemd:

```
systemctl status sshd
● sshd.service - OpenSSH server daemon
   Loaded: loaded
(/usr/lib/systemd/system/sshd.service; enabled;
vendor preset: enabled)
   Active: active (running) since Mon 2019-01-21
13:37:25 EST; 1 months 24 days ago
     Docs: man:sshd(8)
           man:sshd_config(5)
 Main PID: 5627 (sshd)
    Tasks: 3
   CGroup: /system.slice/sshd.service
           ├─ 5627 /usr/sbin/sshd -D
           ├─24349 sshd: [accepted]
           └─24350 sshd: [net]
```

A Systemd “Unit” File:

/usr/lib/systemd/system/sshd.service

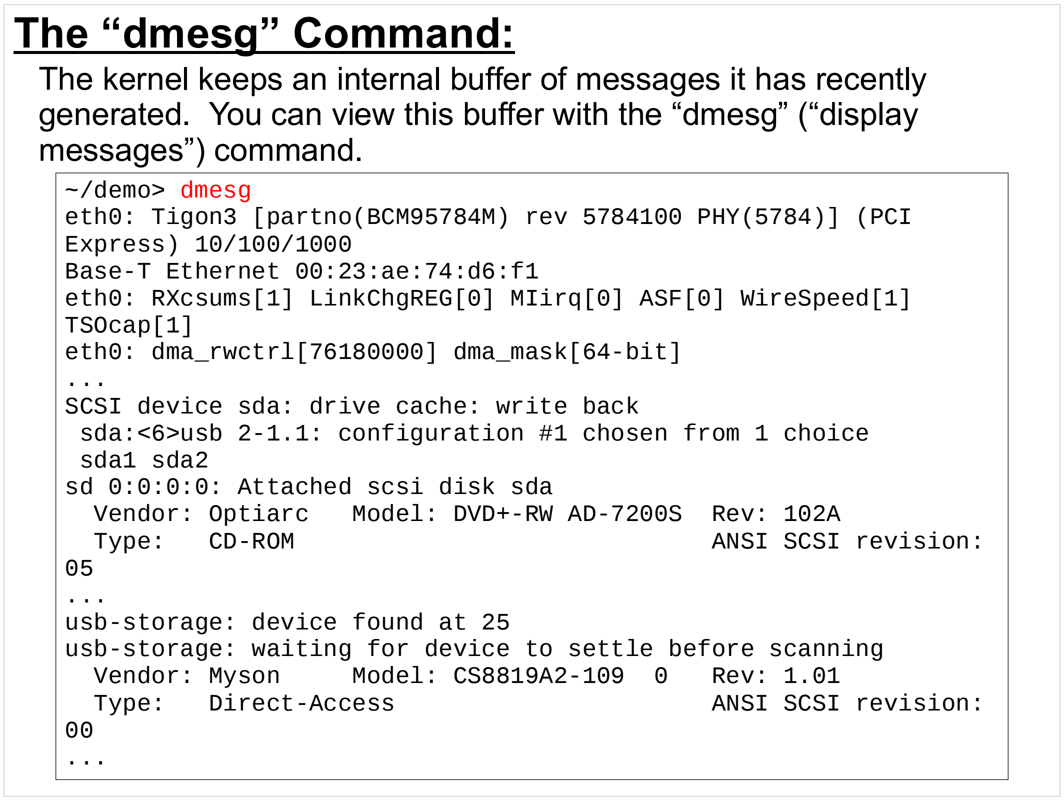
```
[Unit]
Description=OpenSSH server daemon
Documentation=man:sshd(8) man:sshd_config(5)
After=network.target sshd-keygen.service
Wants=sshd-keygen.service

[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/sshd
ExecStart=/usr/sbin/sshd -D $OPTIONS
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
RestartSec=42s

[Install]
WantedBy=multi-user.target
```

Part 3: System Logging

iptables and most services will generate warnings or error messages when appropriate. Here's a little information about where to find these messages.



The “dmesg” Command:

The kernel keeps an internal buffer of messages it has recently generated. You can view this buffer with the “dmesg” (“display messages”) command.

```
~/demo> dmesg
eth0: Tigon3 [partno(BCM95784M) rev 5784100 PHY(5784)] (PCI
Express) 10/100/1000
Base-T Ethernet 00:23:ae:74:d6:f1
eth0: RXcsums[1] LinkChgREG[0] MIirq[0] ASF[0] WireSpeed[1]
TSOcap[1]
eth0: dma_rwctrl[76180000] dma_mask[64-bit]
...
SCSI device sda: drive cache: write back
 sda:<6>usb 2-1.1: configuration #1 chosen from 1 choice
 sda1 sda2
sd 0:0:0:0: Attached scsi disk sda
  Vendor: Optiarc   Model: DVD+-RW AD-7200S  Rev: 102A
  Type:   CD-ROM                             ANSI SCSI revision:
05
...
usb-storage: device found at 25
usb-storage: waiting for device to settle before scanning
  Vendor: Myson     Model: CS8819A2-109   0   Rev: 1.01
  Type:   Direct-Access                      ANSI SCSI revision:
00
...
```

The messages shown by dmesg are in a “ring buffer” in kernel memory. This is a fixed-length storage area that wraps back to the top whenever the bottom is reached. So, dmesg will only show you the most recently-issued messages.

As you can see above, you can find out some useful information about the system by looking at the output of dmesg.

System Log Files:

System messages are also stored in log files, typically in the directory /var/log. These are plain text files. Each line begins with a date and time, followed by the name of the computer that generated the message (usually the local computer). The format of the rest of the line is left up to whatever program generated that particular message. The names of the log files will vary from one distribution to another:

Red Hat/Fedora/CentOS:

- /var/log/messages
- /var/log/secure
- /var/log/maillog

Ubuntu:

- /var/log/messages
- /var/log/syslog
- /var/log/auth.log
- /var/log/mail.*
- /var/log/kern.log

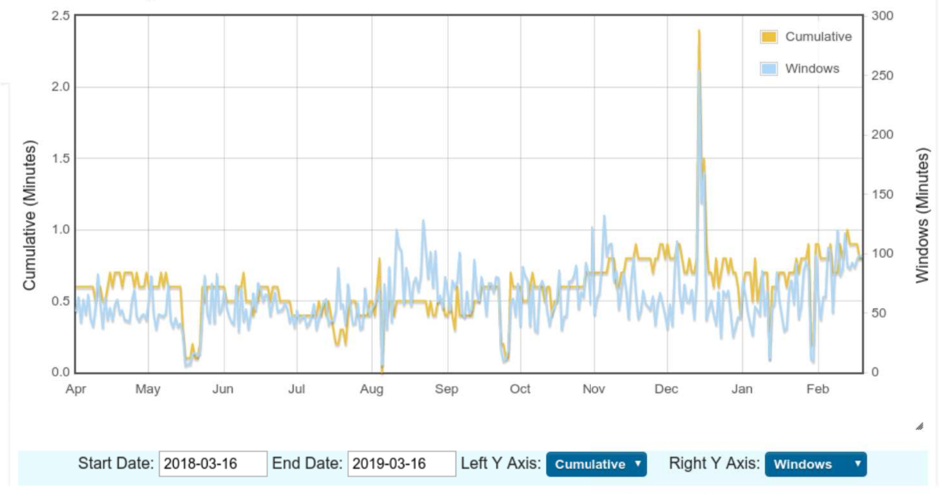
A standard facility, called “syslog”, exists for logging messages. Programs aren't required to use it, but many do. The location of log files controlled by syslog is determined by the file /etc/syslog.conf.

Part 4: Auditing Security

How do you know if your computer is secure? Here are some tools to help.

The Problem:

Today, any computer on the Internet is under constant attack. The graph below, created by the Internet Storm Center (ISC), which tracks malicious traffic on the internet, shows the mean time, in minutes, between attacks on a given computer. They call this the "survival time", since there's a good chance that a computer with an unpatched security vulnerability will be broken into by one of these attacks.



Start Date: 2018-03-16 End Date: 2019-03-16 Left Y Axis: Cumulative Right Y Axis: Windows

https://isc.sans.edu/survivaltime.html

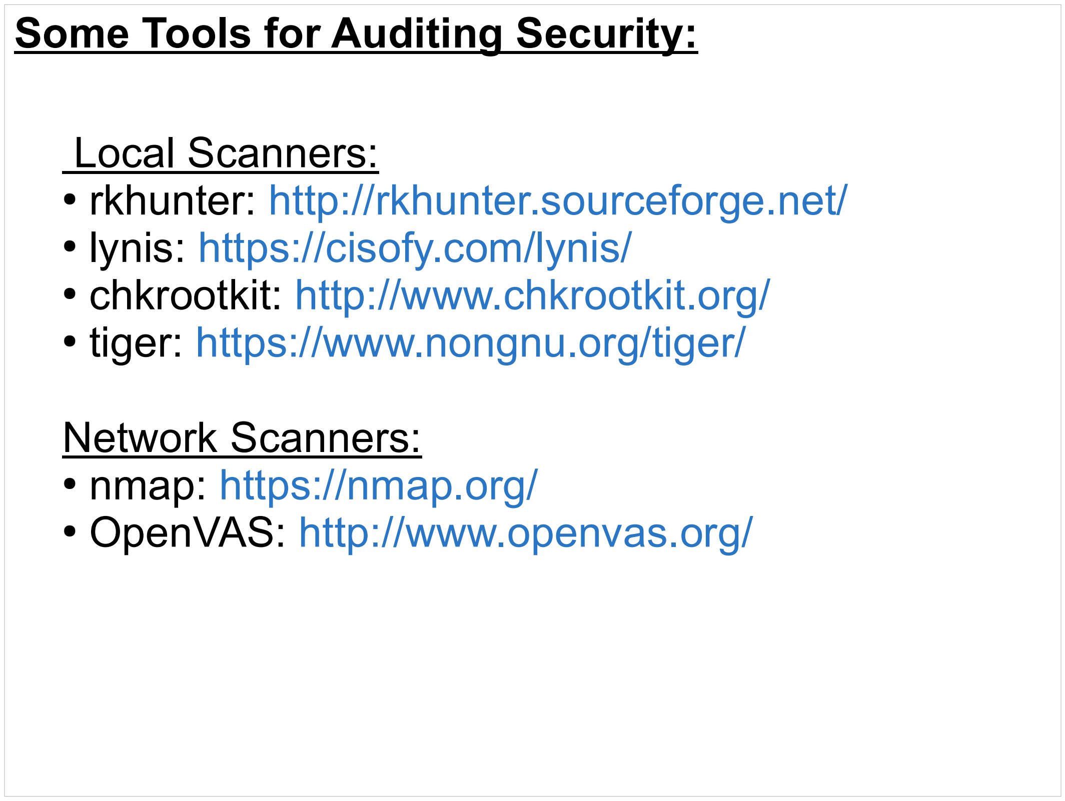
We tend to think of our computers as relatively safe from intrusion, but as you can see, they're under constant attack. There are 3.9 billion Internet users, and each of them has equal access to your computer. It's as easy for a hacker in Hong Kong to break into your computer as it is for him to break into his next-door neighbor's computer (and it's probably much less likely that he'll be prosecuted).

The attacks on your computer are automated. One hacker can simultaneously try breaking into thousands of computers, with very little effort. And even if most computer users are friendly, with 3.9 billion users worldwide, from there are many who aren't.

Imagine that your house was surrounded 24 hours a day by burglars trying to break in at every door and window. How long would it be before one of them succeeded? That's the situation we have now with computers connected to the Internet.

Security Principles:

- Don't run services you don't need.
- Use firewall rules that only allow as much network access as you need.
- Use difficult passwords, and write them down if you need to.
- Keep up with software updates.
- Only log in as the “root” user when you need to, and only for as long as you need .



Some Tools for Auditing Security:

Local Scanners:

- rkhunter: http://rkhunter.sourceforge.net/
- lynis: https://cisofy.com/lynis/
- chkrootkit: http://www.chkrootkit.org/
- tiger: https://www.nongnu.org/tiger/

Network Scanners:

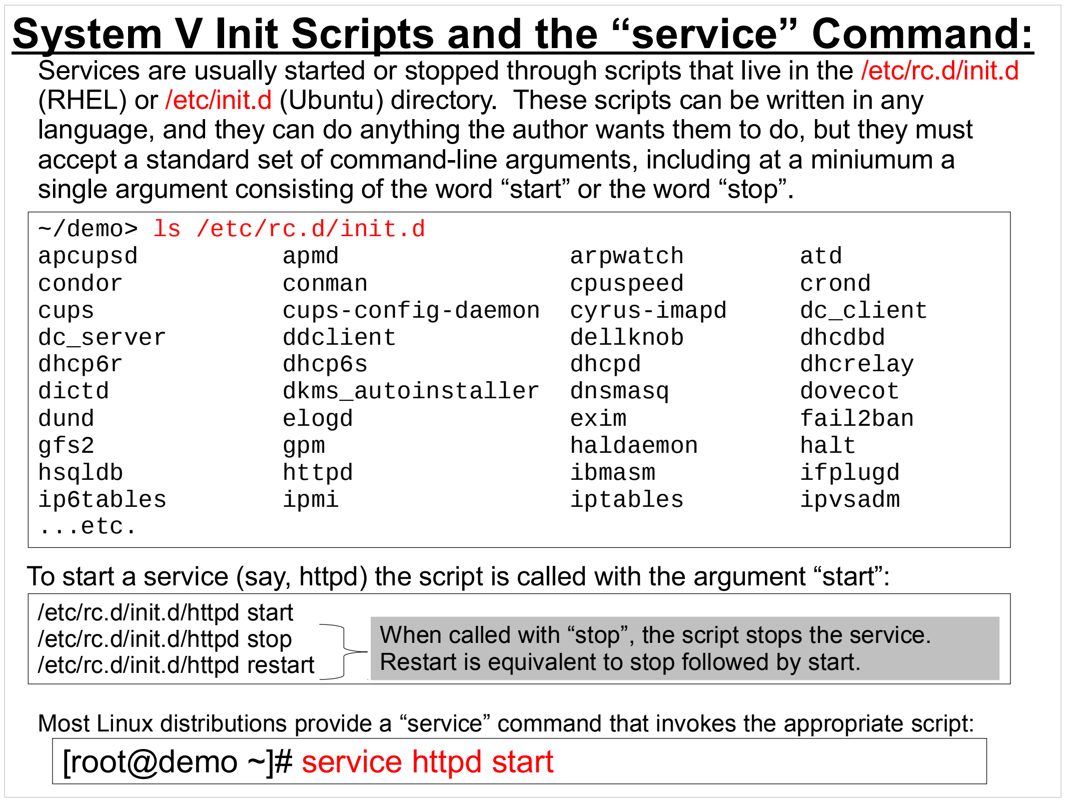
- nmap: https://nmap.org/
- OpenVAS: http://www.openvas.org/

The local scanners above are all good tools. Each of them will give you slightly different recommendations, but all of them are very verbose. You'll need to think about their results, and consider whether their recommendations are right for you.

The nmap scanner can be used to scan for open ports on computers on your local network. It's a good tool for checking the configuration of your computers.

OpenVAS is a powerful tool that scans for known security vulnerabilities on your network. It's optionally available as a ready-to-use virtual machine image. OpenVAS is the open-source descendant of the (previously open-source) proprietary Nessus scanning system from Tenable.

Thanks!



System V Init Scripts and the “service” Command:

Services are usually started or stopped through scripts that live in the /etc/rc.d/init.d (RHEL) or /etc/init.d (Ubuntu) directory. These scripts can be written in any language, and they can do anything the author wants them to do, but they must accept a standard set of command-line arguments, including at a miniumum a single argument consisting of the word “start” or the word “stop”.

```
~/demo> ls /etc/rc.d/init.d
apcupsd          apmd                 arpwatch        atd
condor           conman               cpuspeed        crond
cups             cups-config-daemon   cyrus-imapd     dc_client
dc_server        ddclient             dellknob        dhcdbd
dhcp6r           dhcp6s               dhcpd           dhcrelay
dictd            dkms_autoinstaller   dnsmasq         dovecot
dund             elogd                exim            fail2ban
gfs2             gpm                  haldaemon       halt
hsqldb           httpd                ibmasm          ifplugd
ip6tables        ipmi                 iptables        ipvsadm
...etc.
```

To start a service (say, httpd) the script is called with the argument “start”:

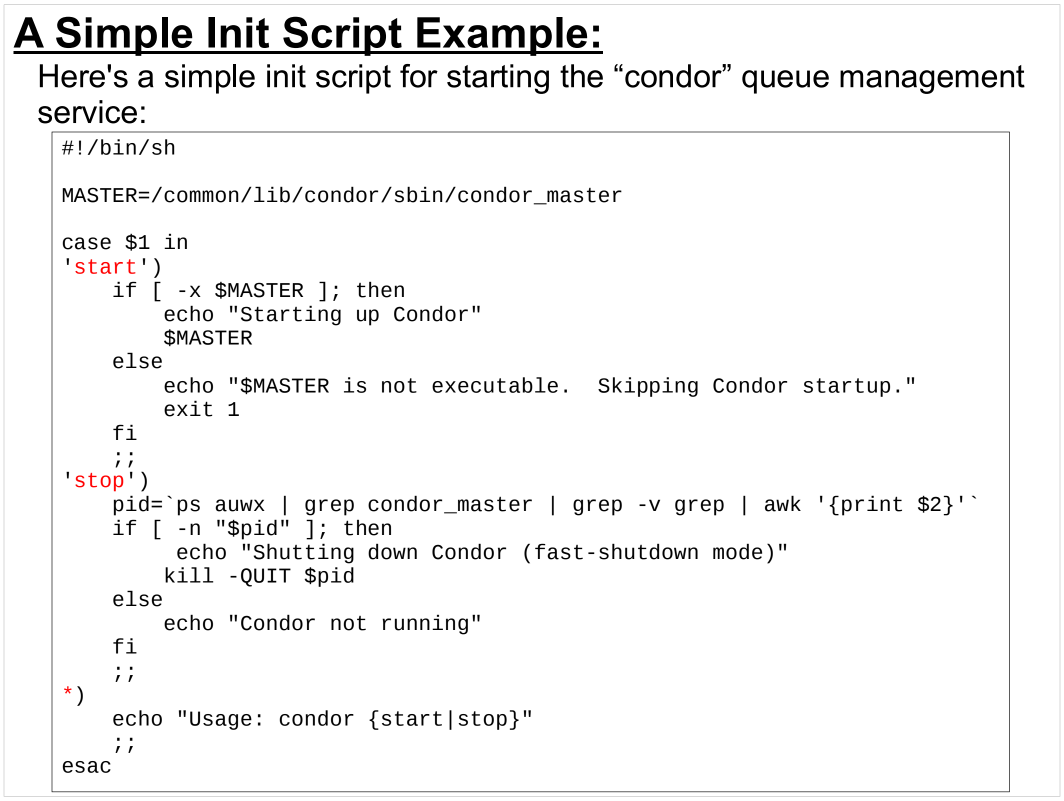
```
/etc/rc.d/init.d/httpd start
/etc/rc.d/init.d/httpd stop
/etc/rc.d/init.d/httpd restart
```

When called with “stop”, the script stops the service.
Restart is equivalent to stop followed by start.

Most Linux distributions provide a “service” command that invokes the appropriate script:

```
[root@demo ~]# service httpd start
```

The “System V” or “SysV” init script standard originally came from AT&T System V Unix.



A Simple Init Script Example:

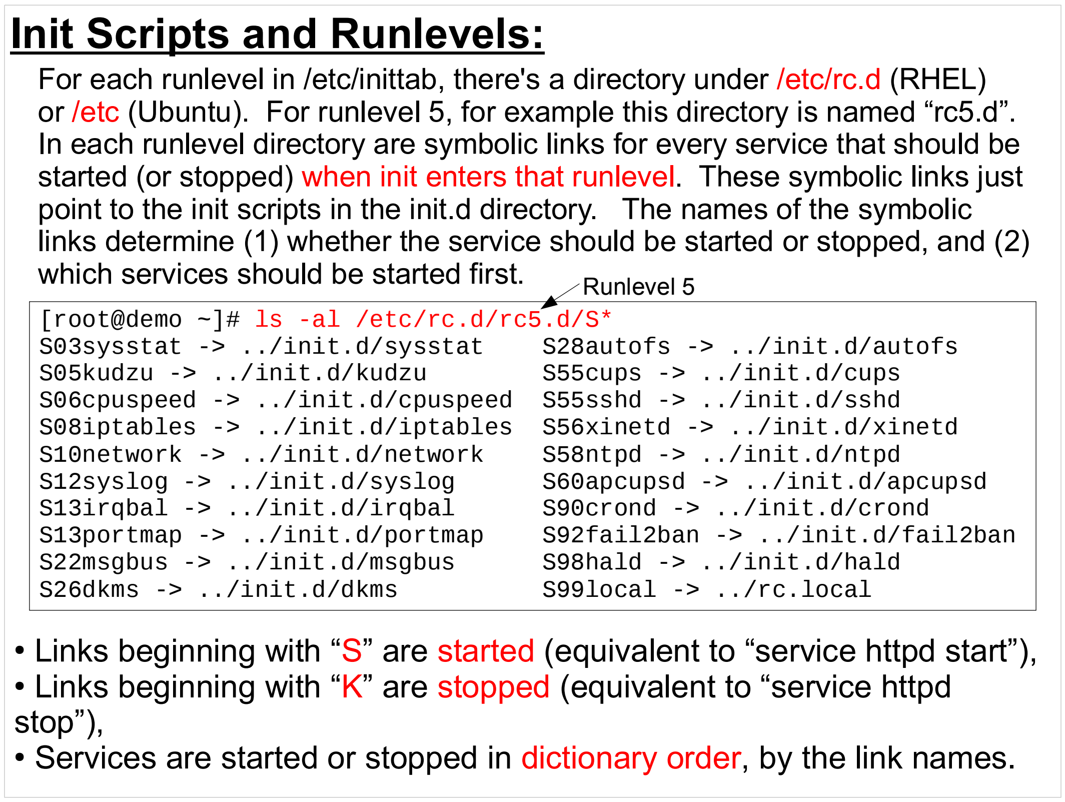
Here's a simple init script for starting the “condor” queue management service:

```
#!/bin/sh

MASTER=/common/lib/condor/sbin/condor_master

case $1 in
'start')
    if [ -x $MASTER ]; then
        echo "Starting up Condor"
        $MASTER
    else
        echo "$MASTER is not executable.  Skipping Condor startup."
        exit 1
    fi
    ;;
'stop')
    pid=`ps auwx | grep condor_master | grep -v grep | awk '{print $2}'`
    if [ -n "$pid" ]; then
         echo "Shutting down Condor (fast-shutdown mode)"
        kill -QUIT $pid
    else
        echo "Condor not running"
    fi
    ;;
*)
    echo "Usage: condor {start|stop}"
    ;;
esac
```

As I said earlier, these scripts can be written in any language, but most of them are Bourne shell scripts. The one above is a simple example, that just uses a “case” statement to decide what to do when the script is given a “start” or “stop” command.



Init Scripts and Runlevels:

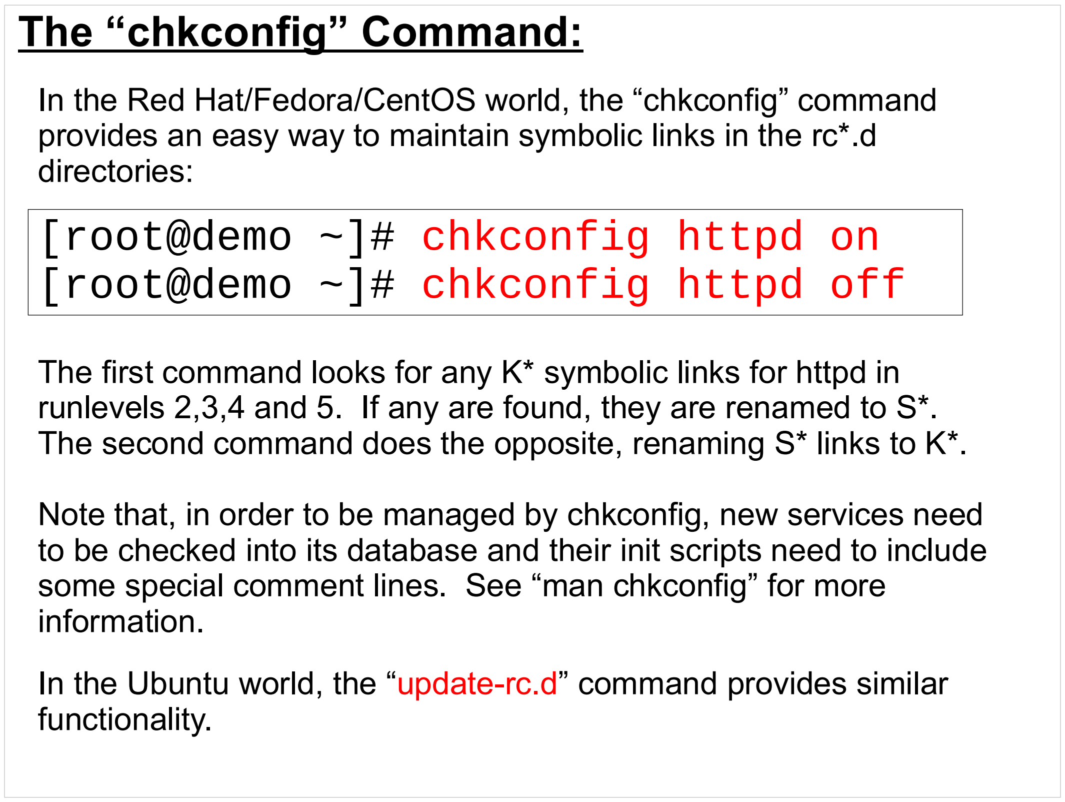
For each runlevel in /etc/inittab, there's a directory under /etc/rc.d (RHEL) or /etc (Ubuntu). For runlevel 5, for example this directory is named “rc5.d”. In each runlevel directory are symbolic links for every service that should be started (or stopped) when init enters that runlevel. These symbolic links just point to the init scripts in the init.d directory. The names of the symbolic links determine (1) whether the service should be started or stopped, and (2) which services should be started first.

Runlevel 5

```
[root@demo ~]# ls -al /etc/rc.d/rc5.d/S*
S03sysstat -> ../init.d/sysstat    S28autofs -> ../init.d/autofs
S05kudzu -> ../init.d/kudzu        S55cups -> ../init.d/cups
S06cpuspeed -> ../init.d/cpuspeed  S55sshd -> ../init.d/sshd
S08iptables -> ../init.d/iptables  S56xinetd -> ../init.d/xinetd
S10network -> ../init.d/network    S58ntpd -> ../init.d/ntpd
S12syslog -> ../init.d/syslog      S60apcupsd -> ../init.d/apcupsd
S13irqbal -> ../init.d/irqbal      S90crond -> ../init.d/crond
S13portmap -> ../init.d/portmap    S92fail2ban -> ../init.d/fail2ban
S22msgbus -> ../init.d/msgbus      S98hald -> ../init.d/hald
S26dkms -> ../init.d/dkms          S99local -> ../rc.local
```

- Links beginning with “S” are started (equivalent to “service httpd start”),
- Links beginning with “K” are stopped (equivalent to “service httpd stop”),
- Services are started or stopped in dictionary order, by the link names.

So, here's how services get started automatically at boot time.



The “chkconfig” Command:

In the Red Hat/Fedora/CentOS world, the “chkconfig” command provides an easy way to maintain symbolic links in the rc*.d directories:

```
[root@demo ~]# chkconfig httpd on
[root@demo ~]# chkconfig httpd off
```

The first command looks for any K* symbolic links for httpd in runlevels 2,3,4 and 5. If any are found, they are renamed to S*. The second command does the opposite, renaming S* links to K*.

Note that, in order to be managed by chkconfig, new services need to be checked into its database and their init scripts need to include some special comment lines. See “man chkconfig” for more information.

In the Ubuntu world, the “update-rc.d” command provides similar functionality.

Note that chkconfig doesn't actually start or stop the service. It just configures the service to start automatically (or not) at boot time. If you want to start or stop a service right now, use the “service” command.

More About /etc/inittab, and rc.local:

When init enters the given runlevel, it executes the program /etc/rc.d/rc with the runlevel as an argument. This program is the thing that looks at the /etc/rc.d/rc*.d directory and executes the appropriate init scripts to start services.

```
l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6
```

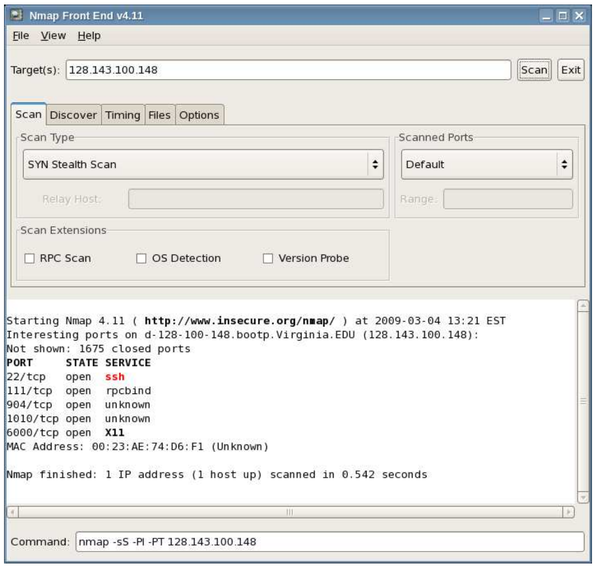
inittab entries to start services.

Also note that, for runlevels 2,3,4 and 5, the rc*.d directories usually contain a symbolic link named “S99local”, pointing to the file /etc/rc.d/rc.local. This script will be one of the last init scripts executed when entering the given runlevel. It's a good place to put miscellaneous commands that you'd like to run each time your computer boots.

```
~/demo>  ls -al /etc/rc.d/rc5.d/S99local
/etc/rc.d/rc5.d/S99local -> ../rc.local
```

Using “nmap” to Audit Security:

Nmap is a “port scanner”. When pointed at a remote computer, it attempts to connect to a each of a set of ports. Based on these connection attempts, nmap determines what network services are listening on the remote computer. Nmap is commonly run through a graphical interface called “nmapfe” (“nmap front end”):



Target(s): 128.143.100.148

```
Starting Nmap 4.11 ( http://www.insecure.org/nmap/ ) at 2009-03-04 13:21 EST
Interesting ports on d-128-100-148.bootp.Virginia.EDU (128.143.100.148):
Not shown: 1675 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
111/tcp  open  rpcbind
904/tcp  open  unknown
1010/tcp open  unknown
6000/tcp open  X11
MAC Address: 00:23:AE:74:D6:F1 (Unknown)

Nmap finished: 1 IP address (1 host up) scanned in 0.542 seconds
```

Command: nmap -sS -PI -PT 128.143.100.148

Nmap can be used to see what services your computer is offering to the outside world.

Nmap is a great tool for looking for security problems on your own computer, but please don't point it at computers you don't own.

Using “fail2ban”:

One of the most common types of malicious activity on the Internet is the “brute-force ssh attack”. In these attacks, Bad Guys use automated tools to try logging into your computer by ssh. They use a dictionary of common usernames and passwords, and they may make thousands of login attempts. In the best case, this uses some of your computer's resources. In the worst case, they stumble upon a valid username/password combination and gain access to your computer.

One of the best tools for dealing with these attacks is “fail2ban”. Fail2ban looks for groups of unsuccessful login attempts and automatically blocks the attacking machine, using iptables firewall rules. Fail2ban remembers which hosts are blocked, and automatically unblocks them after some timeout period.

/var/log/fail2ban.log:

```
2009-03-03 10:28:31,776 fail2ban.actions: WARNING [ssh-iptables] Ban 85.233.64.178
2009-03-03 10:38:31,986 fail2ban.actions: WARNING [ssh-iptables] Unban 85.233.64.178
2009-03-03 13:31:18,984 fail2ban.actions: WARNING [ssh-iptables] Ban 195.14.29.12
2009-03-03 13:41:19,264 fail2ban.actions: WARNING [ssh-iptables] Unban 195.14.29.12
2009-03-03 13:45:47,325 fail2ban.actions: WARNING [ssh-iptables] Ban 195.14.29.12
2009-03-03 13:55:47,555 fail2ban.actions: WARNING [ssh-iptables] Unban 195.14.29.12
2009-03-04 06:49:17,178 fail2ban.actions: WARNING [ssh-iptables] Ban 116.7.255.86
2009-03-04 06:59:17,421 fail2ban.actions: WARNING [ssh-iptables] Unban 116.7.255.86
2009-03-04 08:35:42,481 fail2ban.actions: WARNING [ssh-iptables] Ban 122.9.63.150
2009-03-04 08:45:42,623 fail2ban.actions: WARNING [ssh-iptables] Unban 122.9.63.150
```

Arno's Iptables Firewall:

“Arno's Iptables Firewall” (AIF) is a script to help automate the creation of a complex set of firewall rules. The script reads a configuration file that describes, at a high level, the layout of the desired firewall. The configuration is usually “/etc/arno-iptables-firewall/firewall.conf”.

AIF can be used for even trivial firewalls, but it's invaluable for setting up complex firewalls with multiple network interfaces, NAT, forwarding, etc.

Here's a tiny section of the firewall rules produced by AIF for a non-trivial configuration:

```
...
-A VALID_CHK -p tcp -m tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG FIN,PSH,URG -m limit --limit 3/min -j LOG --log-prefix "Stealth XMAS scan: " --log-level 7
-A VALID_CHK -p tcp -m tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG FIN,SYN,RST,ACK,URG -m limit --limit 3/min -j LOG --log-prefix "Stealth XMAS-PSH scan: " --log-level 7
-A VALID_CHK -p tcp -m tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG FIN,SYN,RST,PSH,ACK,URG -m limit --limit 3/min -j LOG --log-prefix "Stealth XMAS-ALL scan: " --log-
level 7
-A VALID_CHK -p tcp -m tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG FIN -m limit --limit 3/min -j LOG --log-prefix "Stealth FIN scan: " --log-level 7
-A VALID_CHK -p tcp -m tcp --tcp-flags SYN,RST SYN,RST -m limit --limit 3/min -j LOG --log-prefix "Stealth SYN/RST scan: " --log-level 7
-A VALID_CHK -p tcp -m tcp --tcp-flags FIN,SYN FIN,SYN -m limit --limit 3/min -j LOG --log-prefix "Stealth SYN/FIN scan(?): " --log-level 7
-A VALID_CHK -p tcp -m tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG NONE -m limit --limit 3/min -j LOG --log-prefix "Stealth Null scan: " --log-level 7
-A VALID_CHK -p tcp -m tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG FIN,PSH,URG -j DROP
-A VALID_CHK -p tcp -m tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG FIN,SYN,RST,ACK,URG -j DROP
-A VALID_CHK -p tcp -m tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG FIN,SYN,RST,PSH,ACK,URG -j DROP
-A VALID_CHK -p tcp -m tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG FIN -j DROP
...
```

AIF can be downloaded here: http://rocky.eld.leidenuniv.nl/

TCP Wrappers:

Before firewall rules, we had “tcp_wrappers”. Tcp_wrappers is a library of functions that helps programs decide on their own whether they will allow a network connection from a particular remote computer. The library, called “libwrap”, provides routines for parsing rules stored in the files /etc/hosts.deny and /etc/hosts.allow, and applying those rules to incoming network connections.

Each line in these files specifies a service and a list of clients (i.e., computers to be allowed or denied access to that service). As a special case, the word “ALL” can be used for either service or client.

The files are processed in this order:

- Access will be granted when a (service,client) pair matches an entry in the /etc/hosts.allow file.
- Otherwise, access will be denied when a (service,client) pair matches an entry in the /etc/hosts.deny file.
- Otherwise, access will be granted.

For example, here are files that allow web server access to everybody, and allow computers at UVa to have access to all services, but deny all other computers access to anything:

hosts.allow

```
httpd: ALL
ALL:  .virginia.edu
```

hosts.deny

```
ALL: ALL
```