CHAPTER 1

Real-Time Network Statistics with Iftop

Monitoring network connections certainly can be frustrating, mainly because they can be established and then disappear within a matter of seconds. In this chapter, I show you how to achieve a Zen-like approach to monitoring network connections on your servers using a command line–based tool called `iftop`. I then finish by walking through the creation of a configuration file that you can use again on different servers once you have your monitoring set up as you prefer.

Monitoring Network Connections with netstat

The `netstat` command-line tool has been a staple among system admins. Although rich in features, including an `auto-refresh` parameter (continuous mode), `netstat` is certainly not designed to do much more than output raw numbers and names (from hosts and ports). To run netstat in continuous mode, for example, you can use:

```
# netstat -c
```

I usually end up running it alongside `watch` to give me the kind of clean screen refreshes I need for different scenarios; for example:

```
# watch -n2 "netstat -tu"
```

In this example, `watch` lets me configure a two-second gap prior to running the command again and updating its output (see Figure 1-1).

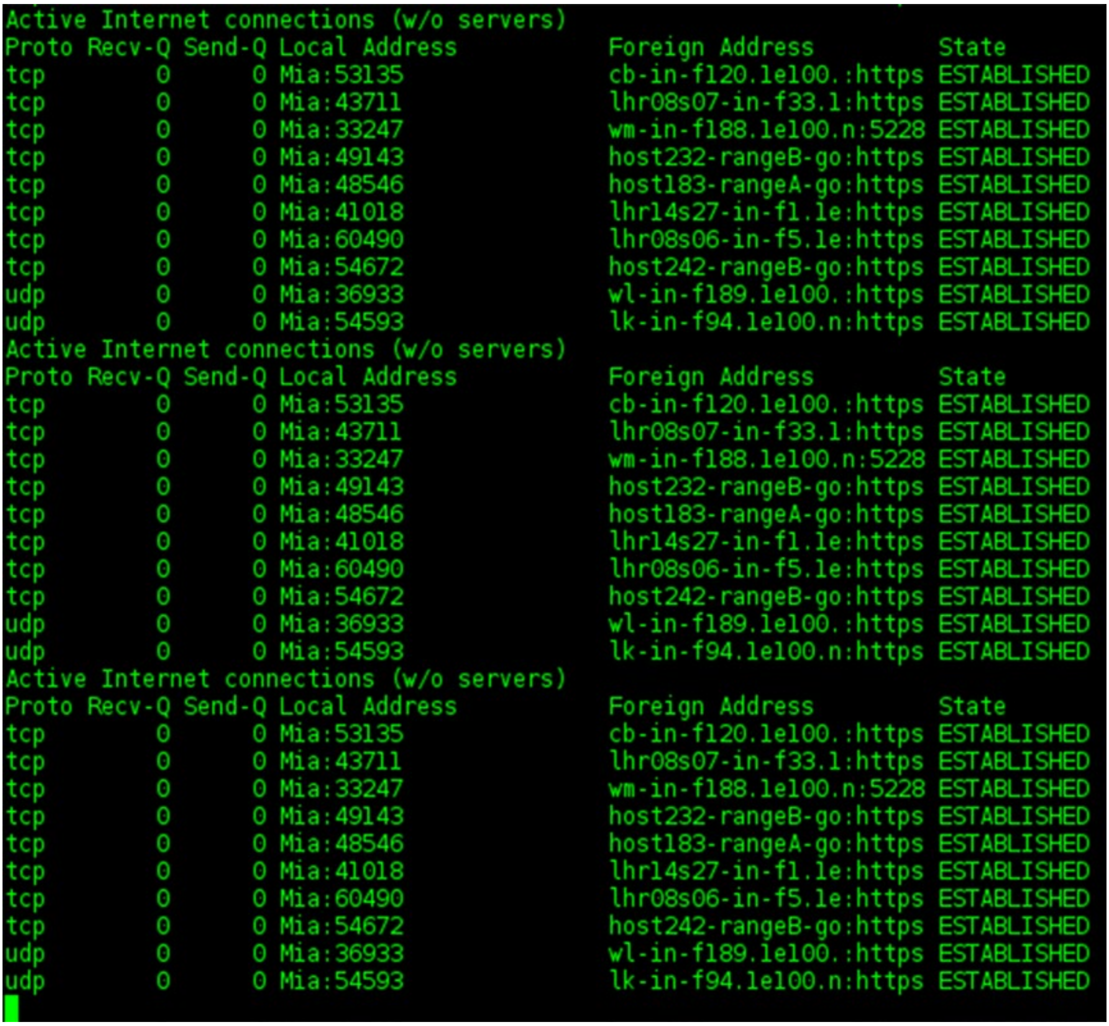
```
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 Mia:35147               host162-rangeA-go:https ESTABLISHED
tcp        0      0 Mia:58928               host227-rangeB-go:https ESTABLISHED
tcp        0      0 Mia:53135               cb-in-f120.1e100.:https ESTABLISHED
tcp        0      0 Mia:39854               lhr08s05-in-f234.:https TIME_WAIT
tcp        0      0 Mia:51598               host226-rangeB-go:https TIME_WAIT
tcp        0      0 Mia:33247               wm-in-f188.1e100.n:5228 ESTABLISHED
tcp        0      0 Mia:49143               host232-rangeB-go:https ESTABLISHED
tcp        0      0 Mia:48546               host183-rangeA-go:https ESTABLISHED
tcp        0      0 Mia:60490               lhr08s06-in-f5.1e:https ESTABLISHED
tcp        0      0 Mia:54672               host242-rangeB-go:https ESTABLISHED
udp        0      0 Mia:57764               lax17s04-in-f3.1e:https ESTABLISHED
udp        0      0 Mia:34254               wa-in-f189.1e100.:https ESTABLISHED
udp        0      0 Mia:36933               wl-in-f189.1e100.:https ESTABLISHED
```

Figure 1-1. *The "watch" command executing "netstat -tu" every two seconds*

In this scenario, the `-tu` switch tells `netstat` to output both TCP and UDP statistics. Using the `watch` option is far slicker than the continuous `-c` parameter, because it adds information to the foot of the last output, although the output is still a little messy and difficult to follow. Incidentally, the following `netstat` command is the one I use the most:

```
# netstat -tulpnc
```

In Figure 1-2, I ask `netstat` to show all local listening ports and then the processes that they belong to. However, `lsof -i`, a command concerned with listing open files, is probably more effective.



```
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 Mia:53135               cb-in-f120.1e100.:https ESTABLISHED
tcp        0      0 Mia:43711               lhr08s07-in-f33.1:https ESTABLISHED
tcp        0      0 Mia:33247               wm-in-f188.1e100.n:5228 ESTABLISHED
tcp        0      0 Mia:49143               host232-rangeB-go:https ESTABLISHED
tcp        0      0 Mia:48546               host183-rangeA-go:https ESTABLISHED
tcp        0      0 Mia:41018               lhr14s27-in-f1.1e:https ESTABLISHED
tcp        0      0 Mia:60490               lhr08s06-in-f5.1e:https ESTABLISHED
tcp        0      0 Mia:54672               host242-rangeB-go:https ESTABLISHED
udp        0      0 Mia:36933               wl-in-f189.1e100.:https ESTABLISHED
udp        0      0 Mia:54593               lk-in-f94.1e100.n:https ESTABLISHED
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 Mia:53135               cb-in-f120.1e100.:https ESTABLISHED
tcp        0      0 Mia:43711               lhr08s07-in-f33.1:https ESTABLISHED
tcp        0      0 Mia:33247               wm-in-f188.1e100.n:5228 ESTABLISHED
tcp        0      0 Mia:49143               host232-rangeB-go:https ESTABLISHED
tcp        0      0 Mia:48546               host183-rangeA-go:https ESTABLISHED
tcp        0      0 Mia:41018               lhr14s27-in-f1.1e:https ESTABLISHED
tcp        0      0 Mia:60490               lhr08s06-in-f5.1e:https ESTABLISHED
tcp        0      0 Mia:54672               host242-rangeB-go:https ESTABLISHED
udp        0      0 Mia:36933               wl-in-f189.1e100.:https ESTABLISHED
udp        0      0 Mia:54593               lk-in-f94.1e100.n:https ESTABLISHED
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 Mia:53135               cb-in-f120.1e100.:https ESTABLISHED
tcp        0      0 Mia:43711               lhr08s07-in-f33.1:https ESTABLISHED
tcp        0      0 Mia:33247               wm-in-f188.1e100.n:5228 ESTABLISHED
tcp        0      0 Mia:49143               host232-rangeB-go:https ESTABLISHED
tcp        0      0 Mia:48546               host183-rangeA-go:https ESTABLISHED
tcp        0      0 Mia:41018               lhr14s27-in-f1.1e:https ESTABLISHED
tcp        0      0 Mia:60490               lhr08s06-in-f5.1e:https ESTABLISHED
tcp        0      0 Mia:54672               host242-rangeB-go:https ESTABLISHED
udp        0      0 Mia:36933               wl-in-f189.1e100.:https ESTABLISHED
udp        0      0 Mia:54593               lk-in-f94.1e100.n:https ESTABLISHED
```

Figure 1-2. *"netstat" output displays changing information by repeating the output at the foot of the screen upon refresh*

Introducing iftop

Thankfully, there's a utility that takes away painful eyestrain: `iftop` (`http://www.ex-parrot.com/pdw/iftop/`). `iftop` is to networks what top is to CPUs. And, in the same way `ifconfig` refers to configuring an interface, the friendly `iftop` stands for "interface top".

The main difference between `iftop` and other command line–based tools is that `iftop` instantly draws highly useful bar graphs (among other graphical options) and I can't emphasize enough how many times it has saved the day when diagnosing an urgent server or network issue.

The fact that there's no pained preamble with device drivers or libraries—`iftop` "just works"—makes all the difference when you're in a hurry. Its small footprint might also be helpful in diagnosing a customer server that doesn't have `iftop` installed: the tiny package can be easily dropped on a memory stick when the networking is broken on the problematic server.

Installing iftop

On Debian-based systems, you can install `iftop` with the following command:

```
# apt-get install iftop
```

On Red Hat derivatives, you can install the EPEL (Extra Packages for Enterprise Linux) RPM, using for example:

```
# rpm -ivh http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm
```

For older versions of the Red Hat family, you can follow the instructions at `http://www.cyberciti.biz/faq/fedora-sl-centos-redhat6-enable-epel-repo/`.

You should then be able to install `iftop` as usual:

```
# yum install iftop
```

If you execute `iftop` with just an interface to examine, you can spawn it using the following command:

```
# iftop -i eth0
```

Figure 1-3 shows a sample output of this command. The output shows a number of remote hosts (including nicknames added to the file `/etc/hosts`, as well as fully qualified domain names, which are really just normal hostnames that I've abbreviated) in the middle column and the local machine named `Sula` in the left column.

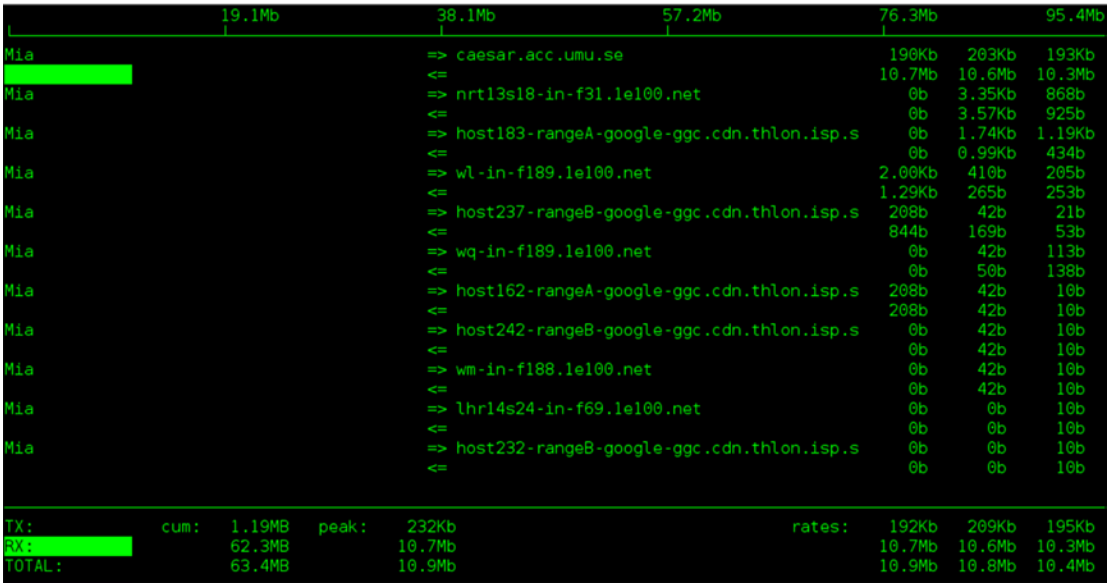


Figure 1-3. *The default "iftop -i eth0" output listening to an interface enabled*

On the right side, you can see three columns. The excellent `iftop` refers to this as its *display order* and the columns deal with different time-delay averages. By default, this appears (at least) to be two-second, ten-second, and forty-second averages. These values can be configured separately, so don't let that confuse you too much initially. In addition, it's easy to change the overall display using those columns by pressing the 1, 2, or 3 keys to respectively filter by the aforementioned 2s, 10s, or 40s averages.

As an aside, a two-second average is really short; I love it having come from a background filled with what felt like lengthy five-minute SNMP averages. I can see very quickly what's just changed on the network and although two seconds isn't real-time, it's very close to it and certainly has its place on today's busy Internet. I find that it's just long enough for you to be able to spot something without worrying about freezing the screen in case you missed it.

When you're running the default config without specifying any options, `iftop` outputs the busiest hosts in the last ten seconds (in other words, by using a ten-second average). It also groups hosts in pairs to choose the busiest *pair* of combined inbound and outbound traffic.

Finally, at the end of the output, you are presented with a number of totals. These include useful statistics, such as the amount of data transferred in megabytes (MB) as well as forty-second averages of traffic, usually in megabits (Mb), but also sometimes in *kb* for kilobytes.

Controlling iftop from the Keyboard

In addition to providing a slick graphical display, even through an SSH terminal, `iftop` lets you modify your configuration at the press of a key. For example, in the course of a sysadmin's work day, you could be checking all sorts of bad networking habits: from monitoring the misconfiguration of a network interface to mitigating a hideously hazardous ARP storm. With `iftop`, you can cycle through a number of options and confidently choose a config parameter to suit your current scenario instantly.

Here are some examples of how `iftop` can make your sysadmin life easier at the press of key:

- To change the source and destination displays, press the *s* key or the *d* key while `iftop` is running. This helps isolate who is sending what, especially if `iftop` is being run on a Linux router (which I'll touch in "Using iftop on Busy Routers" later in this chapter) and forwarding traffic.
- To quickly see which ports are in use, press the *p* key. You can also use the Shift+S and Shift+D keys to expose source and destination ports, respectively. Figure 1-4 demonstrates how friendly `iftop` is with its options and how it dutifully reports, in the top-left of the screen, the result of the keypress that it has just received.
- To cycle through several different displays (similar to horizontal bar graphs), press the *t* key.

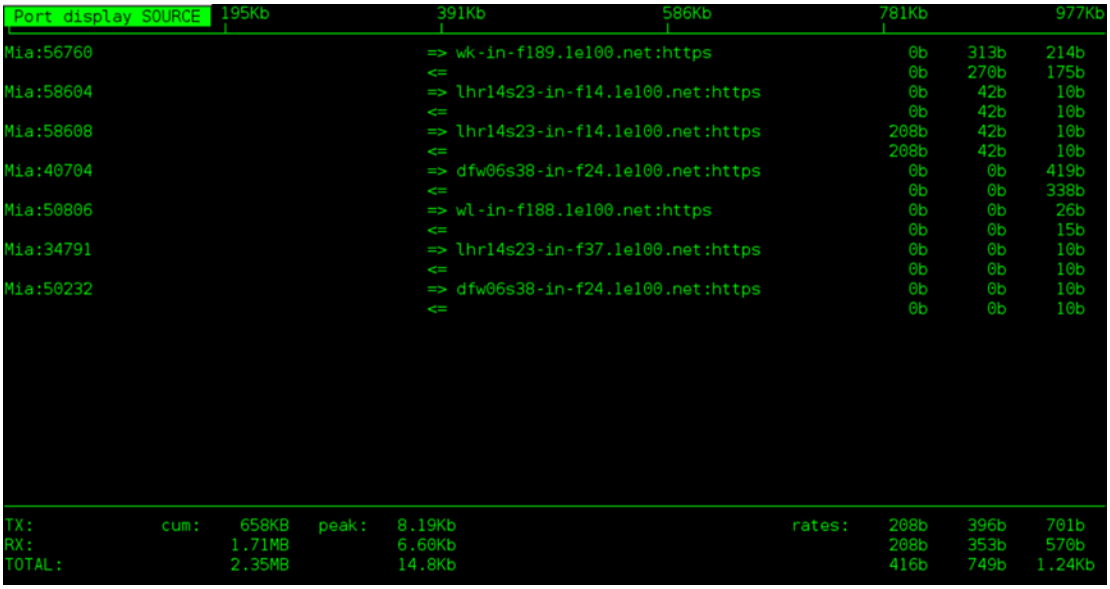


Figure 1-4. *This output shows all the ports that are visible and an option selection, top-left*

Earlier, in Figure 1-3, you saw the standard two-line display. Its purpose might not be entirely obvious however because there's little-to-no traffic moving between hosts (so the white bars, like a bar graph, aren't present), but you can see the => and <= symbols depicting the direction of the traffic.

Figure 1-5 shows a nice feature that is useful in some circumstances. The single line-per-host display (showing both <= and => directions on one line) means that it's easy to spot who is hogging your bandwidth. Figure 1-6 informs you who is moving the most outbound traffic with a single => directional pointer. These two options (the aggregation of inbound and outbound traffic onto one line and choosing a particular direction that you're interested in) are best suited for very busy networks.

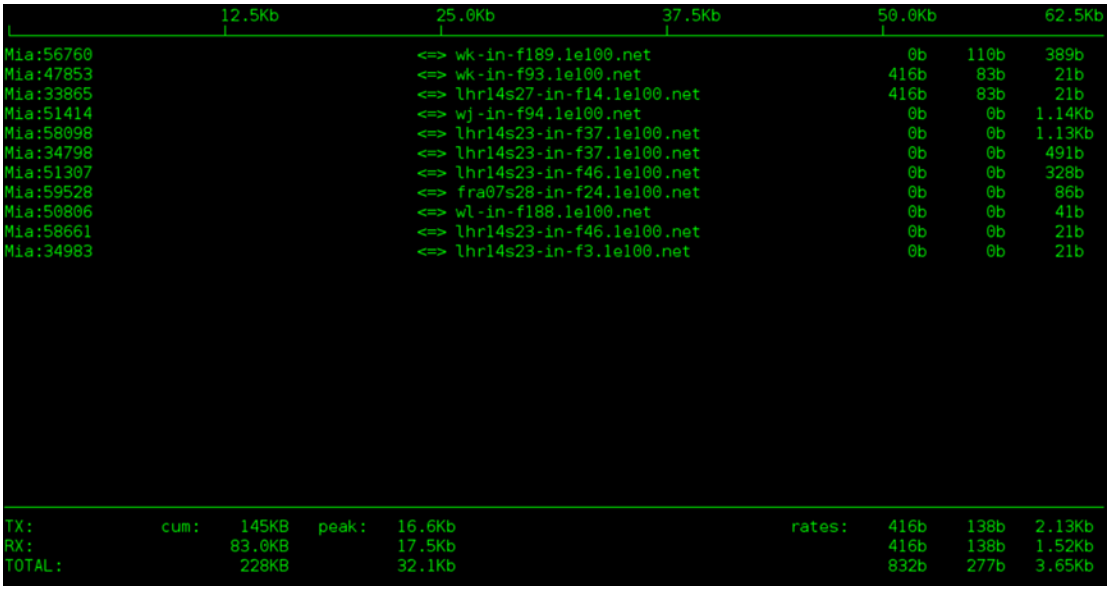


Figure 1-5. *Traffic is displayed with both hosts occupying one line*

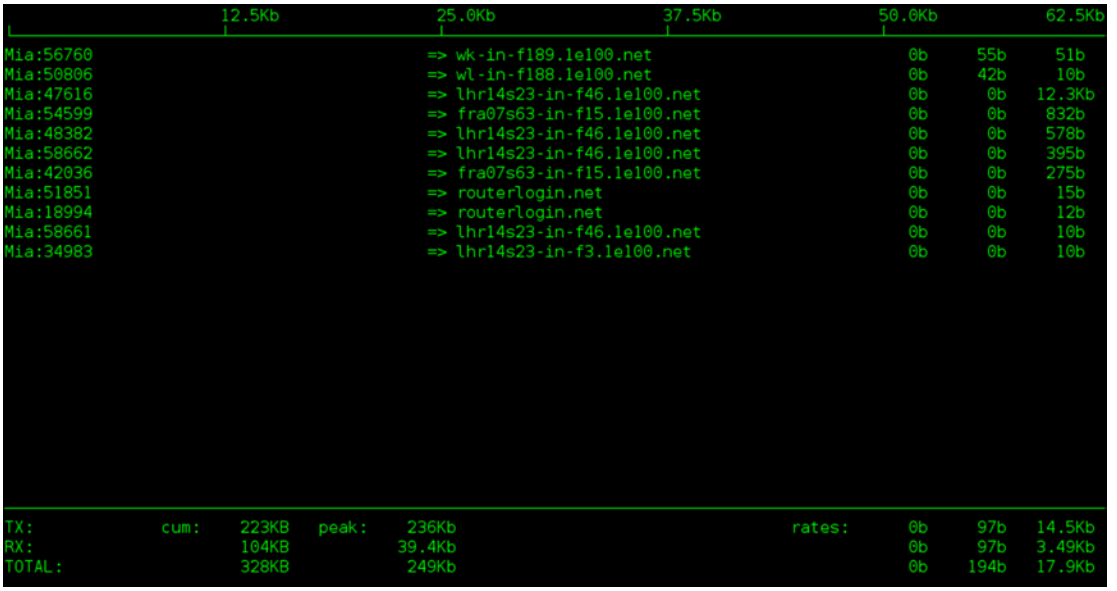


Figure 1-6. *Here "iftop" shows "sent" traffic only. The next selectable option is "received" traffic only*

Adding Screen Filters

The purpose of the *screen filter* is to tidy up a screen that is far too busy to keep track of. Imagine, for example, that you check your network connections and suddenly spot a host that looks suspicious. In that case, quickly press *P* to pause (or freeze) the ever-changing display and then the *l* key, followed by the hostname, which you've just spotted to add a screen filter. That way, the trusty `iftop` only passes back the connection information for that host on its own for closer inspection.

Figure 1-7 shows `iftop` asking for the *screen filter* phrase or keyword to match, in the top left of the screen. If it's a tricky IP address or long hostname, then you should be able to cut and paste into that field.

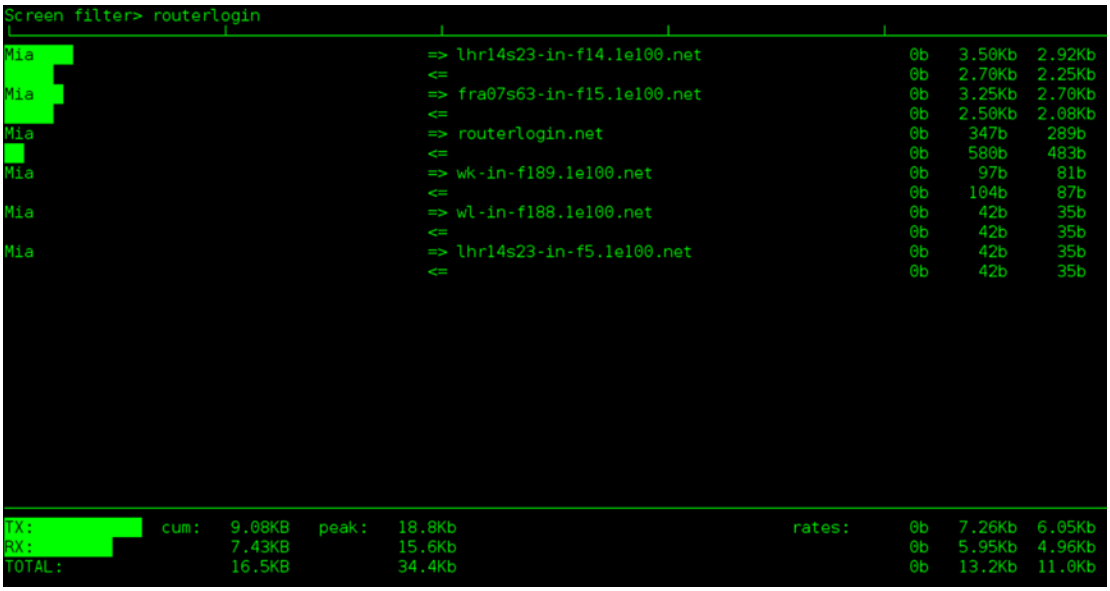


Figure 1-7. *Filter that screen output to make more sense of it with a "screen filter"*

One key difference between other filters and a screen filter is that the totals at the bottom of the screen are not affected by any screen-filtering functionality.

Using Regular Expressions

`Iftop` also supports regular expressions, otherwise known as regex, following the POSIX format. The man page offers an excellent filter, if not one that's slightly more complicated than others, to get you started with some of the ways that you can manipulate `iftop`:

```
not ether host ff:ff:ff:ff:ff:ff
```

This very useful filter ignores all broadcast packets. Such a filter, like all others incidentally, can equally be applied as a screen filter with the letter *l* or the letter *f* as a live filter change to the currently enabled net filter. Alternatively, you can add it on the command line at launch time as well as in the start-up configuration file (which I will look at further in a moment). By removing broadcast traffic, you are effectively sorting the wheat from the chaff so that you can focus on the connections on your server directly and not the normally non-trivial amount of network noise filling your screen up with connections.

`iftop` also accepts regular expressions in the same way that `ngrep` or `tcpdump` do; for example:

```
port smtp and host mail.chrisbinnie.tld
```

If I wanted to monitor traffic to my mail server but avoid all SMTP traffic (which uses TCP port 25), I could use the `and` and `not` operators like this:

```
host mail.chrisbinnie.tld and not port 25
```

■ **Note** As you're probably aware, Linux will look up the file `/etc/services` to match port numbers against service names. If you don't know which port number a service uses, you can query that file to find out.

As mentioned, these filters can be run in a variety of ways, including from the command line as follows at execution time, saving you typing time with the following syntax:

```
# iftop -f "host mail.chrisbinnie.tld and not port smtp"
```

This startup configuration simply checks for all traffic (in and out) on a mail server that isn't coming or going via TCP port 25.

Other clever filter tricks include "from" and "to" style commands where you can ignore entire directions of traffic at the touch of a key. These take the slightly more professional terminology of `src` and `dst` for *source* and *destination*.

One of my favorite parts of `iftop`'s extensible configuration is the fact that you can mix and match almost every filter with other filters to get you the precision you require. Of course, mistakes (a typo or a missing dot to separate the octets of an IP address) happen, but generally I find the syntax of `iftop` is exceptionally intuitive.

Here are some more examples. You can conjure up a mix of hosts and ports with directional filters as follows:

```
dst host router.chrisbinnie.tld and src port bgpd and not host ntp.chrisbinnie.tld
```

Also, sometimes certain ranges of ports are useful. I can't help but think of `ngrep` and `tcpdump` again in relation to this next option.

The port numbers included in the following example are variously referred to as privileged ports, power ports, or special ports. They are referred to as such because, by default, on Linux systems only root can bind to these ports, leaving the upper level of ephemeral ports to non-superusers. The lowest 1024 TCP ports shown next can be checked as follows for suspicious traffic.

```
src host web.chrisbinnie.tld portrange 0-1024
```

You can also check for a specific protocol (a little like the `ether` option shown previously, but this time a transport protocol as opposed to a link type) using this:

```
ip proto name-of-protocol
```

To monitor a single subnet if your server can see multiple subnets, whether local or remote, use the following command:

```
# iftop -F 10.10.10.0/24
```

■ **Note** It's important to note that an uppercase *F* precedes a network option, as opposed to the lowercase version for enabling certain filtering.

Listening for Network Traffic

I once spent a great deal of time checking network anomalies only to discover that one subnet was effectively "bleeding" traffic over into another. The tool that lead me to this discovery was, of course, `iftop`. The following scenario will help explain what I'm talking about.

Like with some other packet sniffers and network monitors, you can (with superuser privileges) configure your network interface to drop into what is called "promiscuous" mode. `iftop` has the ability to listen for traffic travelling across the network port to which the interface is connected (whether that be a network switch, hub, or otherwise) but that isn't destined for the IP addresses bound to that interface. In other words, in some cases, you can pick and choose what to ignore from traffic that is not actually going to your server but is instead going via your network card. You can try to find out with the `-p` flag like this on the interface called `eth0`:

```
# iftop -p -i eth0
```

And, for personal preferences, the `-B` for viewing bytes rather than the network convention of bits (Kbps) can be added for ease:

```
# iftop -i eth0 -B
```

There are also a number of other display options that you might want to experiment with in order to change the screen to your preference. This switches off bar graphs (which I favor usually for clarity).

```
# iftop -b
```

Two other parameters that may come in handy, `-P` and `-N`:

- The uppercase `-P` option (not the same as *P* for freezing the display) ensures that all the ports involved in a network transaction (source and destination) will be visible in the display.

■ **Note** If higher-end port numbers don't have corresponding entries in the file `/etc/services`, then you'll just see the port number directly. Otherwise, you'll be presented with a port name such as `www` or `domain` for DNS, etc. Incidentally, I never turn this functionality off unless my display is brimming with information.

- The `-N` flag just reverses that option, thereby ignoring `/etc/services` entirely and the associated names, and purely concentrates on outputting raw port numbers only.

Changing Filters Can Take Time

Note that depending on your setup and circumstances, changing real-time filters may not yield immediate noticeable results. This is because like many measurements you're relying on averaging a number of results, albeit within a minuscule window usually, so you essentially have to wait for other parts of the display or network filter to catch up.

Configuring iftop Scenarios

Once you've cut your teeth on some of the more standard `iftop` config options, you might find yourself using it in a number of different scenarios. And, in, and around these scenarios if repetition creeps in then, as mentioned, it's useful to have a set configuration ready to use at the press of a key or two.

For example, you could have a `bash` alias or a simple script that references a few different config files held in the `root` user's home directory for `iftop`. Then you could trigger the pertinent config file by typing something akin to a shortcut for a config file along the lines of:

```
# iftop broadcast storm
```

The default config file referenced inside the home directories (usually `root`'s because its system privileges are needed) is `.iftoprc`.

As promised, I wanted to walk through a few configuration options before passing on a fairly generic configuration file that you can add your own filters to; here goes:

```
# cat /root/.iftoprc

dns-resolution: yes
port-resolution: yes
show-bars: yes
promiscuous: yes
port-display: on
hide-source: no
hide-destination: no
use-bytes: no
sort: 2s
line-display: one-line-both
show-totals: yes
log-scale: yes
```

Many of these configuration options speak for themselves. The option `line-display` lets you combine the total traffic sum of inbound and outbound traffic to immediately show the busiest hosts in the display. This is instead of showing one output for inbound traffic and one for outbound. This `config` option can be changed in a number of ways: two-line, one-line-sent, one-line-received.

You may also want to add the "real" network capacity that you're connected to with the `max-bandwidth` option. Imagine for a moment that you are connected to a 1024Mbps network link (1Gbit), but your external Internet capacity is actually only 100Mbps (this could be your *committed data rate*). Sometimes your LAN might show you bursts of traffic that simply don't add up. In other words, you might see more than 100Mbps going out to the Internet, which simply can't be an accurate reflection of your traffic flows.

The excellent `iftop` lets you specify a bandwidth cap so that you can have a more realistic maximum is shown. One caveat; this needs to be in bits, not bytes:

```
max-bandwidth: 100M
```

Here, you can adjust the ceiling down to 100Mbps, even though you're using a gigabit LAN connection.

Using iftop on Busy Routers

I've successfully used `iftop` on very busy Linux routers (pushing hundreds of Mbit of traffic but more importantly with hundreds of thousands of concurrent network connections) and it stood up admirably time and time again. It's worth emphasizing this point because some other tools crashed the router under a decent level of load. `iftop` continues to serve honorably and remains stable even under high levels of stress. Of course, your mileage may vary, depending on versions and flavors, among other things, so try it in busy development environments first.

Should you try `iftop` on a busy router, it may not immediately occur to you that you will generate a massive amount of DNS lookups if that option is enabled. It can add a fairly noticeable load to a machine if it suddenly receives tens of thousands of tiny DNS packets, so initially for testing at the very least, and especially during busy spells, it's wise to disable them. You can use `-n` at startup to avoid your network interfaces panicking at the sudden, massive packets-per-second influx. It's an easy mistake to make and not one that I'd recommend accidentally doing on a production machine.

Summary

I trust this overview of the supercharged utility that is `iftop` has whet your appetite enough to fire it up and have a go at using it yourself. I have barely scratched the surface of the options available to you. The simple fact that this amazing utility complies with relatively complex filter changes while it's still running (at near real-time) makes it truly outstanding. However, coupled with the myriad of configurable options, suitable for almost all circumstances, it is undeniably a formidable network tool to be reckoned with. And what with the introduction of *regular expressions*, there genuinely is little that you can't observe on a network link with `iftop`. I hope you enjoy putting it through its paces as much as I frequently do.

CHAPTER 11

Nattier Networks

If you're a sysadmin who faces networking challenges on a daily basis, there's a reasonable chance that you have picked up the essentials as well as a few extras along the way. Don't get me wrong, it's absolutely fine to only learn what you need to if the majority of your day is spent software patching and not concerned with networking.

In this chapter, I intend to avoid the full "Networking 101" explanation of how, what, why, and where but instead extol the virtue of the `ip` command to help you pick up some more of the basics. I'll begin with a review of the older networking tool that many sysadmins used first, `ifconfig`.

Old School: Using ifconfig

The first tool I was exposed to for checking which IP address was in use on a server (or a Linux machine of any variety), was `ifconfig`. To find out which IP address is bound on each interface, I used:

```
# ifconfig -a
```

Make no mistake, the `ifconfig` utility comes a close second in ruling the networking landscape, but ultimately falls short behind the superior `ip`. Listing 11-1 provides a list of its supported hardware types for example.

Listing 11-1. The ifconfig Utility Is No Slouch Itself and Supports All Sorts of Network Types

```
loop (Local Loopback) slip (Serial Line IP) cslip (VJ Serial Line IP)
slip6 (6-bit Serial Line IP) cslip6 (VJ 6-bit Serial Line IP) adaptive
(Adaptive Serial Line IP)
strip (Metricom Starmode IP) ash (Ash) ether (Ethernet)
tr (16/4 Mbps Token Ring) tr (16/4 Mbps Token Ring (New)) ax25 (AMPR AX.25)
netrom (AMPR NET/ROM) rose (AMPR ROSE) tunnel (IPIP Tunnel)
ppp (Point-to-Point Protocol) hdlc ((Cisco)-HDLC) lapb (LAPB)
arcnet (ARCnet) dlci (Frame Relay DLCI) frad (Frame Relay Access Device)
sit (IPv6-in-IPv4) fddi (Fiber Distributed Data Interface) hippi (HIPPI)
irda (IrLAP) ec (Econet) x25 (generic X.25)
infiniband (InfiniBand)
```

The `ifconfig` utility's manual offers some useful details, including that it gleans its information from these files presented to the system by the pseudo-filesystem `/proc`:

```
/proc/net/socket
/proc/net/dev
/proc/net/if_inet6
```

Sadly, however, without having to squint in the slightest, at the top of the `ifconfig` utility's manual the "NOTE" section laments:

```
"This program is obsolete!  For replacement check ip addr and ip link. For statistics use ip
-s link".
```

Indeed, the `ip` command is both beautifully short to type and brimming with functionality.

New School: Using ip

Consider the `ip` equivalent of the `ifconfig -a` command:

```
# ip -s link
```

Listing 11-2 shows the output from that short command.

Listing 11-2. Useful Statistics from the ip -s link Command

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    RX: bytes  packets  errors  dropped overrun mcast
    3504956    55996    0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    3504956    55996    0       0       0       0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether 00:30:14:2d:1a:ec brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun mcast
    770413813  44919870 0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    1762383491 5460232  0       0       0       0
```

Incidentally, you can just look at the network interface `eth0` with this command, as opposed to viewing all the interfaces using the previous one:

```
# ip -s link show dev eth0
```

The first element that my eye is drawn to in Listing 11-2 is the individual `RX` (received) and `TX` (transmitted) errors. This is thanks to the fact that I had all sorts of fun and games with Network Interface Card (NIC) drivers misbehaving in the past.

Generally speaking, however, when you push an inordinate amount of traffic through any NIC you can expect a few errors. A novice may just think I'm talking about the "errors" column, but actually I'm including "`errors dropped overrun`" for `RX` and "`errors dropped carrier collsns`" for `TX`. The latter TX statistics are for collisions (which is when two network interfaces on the same network try to send data at the same time and packets are dropped as a result).

Probably most important of all is the word UP on the top line in Listing 11-2 shown under the eth0 interface. This means the interface is enabled.

Next, I'll look at some of the many options that the ip command offers.

Old versus New

As I've mentioned, I was used to adding/checking/changing IP addresses with the ifconfig command previously. It might be useful to check against the equivalent command syntax with the newer utility and the ifconfig command. Here is ifconfig's syntax for adding an IP address:

```
# ifconfig eth0 10.10.2.125 netmask 255.255.255.0 broadcast 10.10.2.255
```

This command applies all three IP address settings to the eth0 interface, each of which can be added or changed individually. You simply remove the other one or two settings to ignore applying them.

The ip command's counterpart looks like this:

```
# ip addr add 10.10.2.125/24 dev eth0
```

This adds the IP address to the interface (and overwrites the settings associated with an existing IP address on that interface if they change). The shorthand version of ip addr is as follows:

```
# ip a
```

This next command provides part of the output of ifconfig -a shown earlier. It can get a little confusing at points, but it's not uncommon to type this shorthand for speed too (with the double a):

```
# ip a a 10.10.2.125/24 dev eth0
```

I've been a little remiss by only implicitly declaring the broadcast address of the IP address and subnet. I was explicit with the ifconfig example, so for the purposes of completion, here it is (added separately, explicitly):

```
# ip a a broadcast 10.10.2.255 dev eth0
```

There is obviously a need for some caution here with the command below, especially if you are logged in remotely over SSH and removing an IP address is as simple as accidentally deleting the IP address that you are connected to.

```
# ip a del 10.10.2.125/24 dev eth0
```

Remember I mentioned the ifconfig utility's "up" and whether that interface was enabled? It's just as easy to alter with the ip command.

```
# ip link set eth0 up
```

And, conversely, down works if you swap it in place of up in order to enable and disable an interface.

Jumbo Frames

Occasionally you need to adjust a network's Maximum Transmission Unit (MTU). Of course, you can break TCP horribly too, but if you're running fast networks (yes, I'm including gigabit networks) then moving the MTU from the bog-standard 1500 bytes to 9000 bytes can be achieved like this to potentially decrease CPU cycles on the networking kit and improve performance:

```
# ip link set mtu 9000 dev eth0
```

You use this command to check that your new MTU value of 9000 bytes has been applied:

```
# ip l sh eth0
```

Here is the output:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9000 qdisc mq state UP qlen 1000
    link/ether 00:42:13:4d:2a:ac brd ff:ff:ff:ff:ff:ff
```

The command `ip l sh eth0` is the shorthand of the following command:

```
# ip link show eth0
```

IPv4 and IPv6

If you're operating in one of those lucky countries to have a decent adoption of IPv6 (I'm pointing the finger here squarely at the UK where, as I'm ashamed to say that we're significantly behind according to Google: https://www.google.com/intl/en/ipv6/statistics.html#tab=per-country-ipv6-adoption), then you can ignore IPv4 and focus on IPv6 interfaces (and vice versa with a "4") as follows:

```
# ip -6 a
```

There's also a very useful `list` command that I'll leave you to guess the shorthand for. To check which interfaces are up, you use:

```
# ip link ls up
```

The route Command

Along the same vein as the now deprecated `ifconfig` utility, the `route` command is a little long-in-the-tooth too. It almost brings a tear to my eye to read this under the "NOTE" section of the `route` command's manual:

```
"This program is obsolete. For replacement check ip route."
```

Fear not, though, as you can do all sorts of things with the `ip` command. Routing is unquestionably an integral part of the `ip` command's raison d'etre.

Embracing all things that mean less typing, you can use the perfectly short `ip r` command in order to check the local routing table of a machine by running the following command:

```
# ip r
```

Here is the output:

```
10.10.0.0/22 dev eth0  proto kernel  scope link  src 10.10.2.34
default via 10.10.0.1 dev eth0
```

The first line tells you that there's a routing entry for the `10.10.0.0` network on `eth0` for the IP address `10.10.2.34`. The second line tells you that if you need to contact any other network, you should forward your requests to the helpful `10.10.0.1` IP address because that's the helpful "default gateway" or "default route".

Consider the `route` command with DNS lookups switched off to lessen the burden on DNS servers and improve your screen's output:

```
# route -n
```

In Listing 11-3, you can see the results from running `route -n` on a very small routing table in this case. Incidentally, at the time of writing there are apparently around "562117" routes held in the global BGP routing table. If you're interested, there's some excellent information here: `http://bgp.potaroo.net`.

Listing 11-3. The Local Routing Table Is Displayed Old School, the Way It Was Meant to Be, Using route -n

```
Kernel IP routing table
Destination     Gateway         Genmask         lags   Metric Ref    Use Iface
10.10.0.0       0.0.0.0         255.255.252.0       U         0 0        0 eth0
0.0.0.0         10.10.0.1       0.0.0.0         UG        0 0        0 eth0
```

It's worth noting that Listing 11-3 isn't volunteering the IP address that corresponds to the `10.10.0.0` network destination, unlike the output from the `ip r` command.

Now that you can view the routing table, let's take a look at manipulating it.

Static Routes

If you don't want to communicate with a network (or IP address) via the default route, you can easily add what's known as a "static route".

■ **Note** A quick reminder for those new to this. The "default gateway" is the "default route" of which I speak. It catches all communication requests that don't have specific routing entries, such as the one that you're about to add by using a default route.

The static route syntax looks like this:

```
# ip route add 154.30.109.0/24 via 10.10.0.4 dev eth0
```

In simple terms, this states that any traffic for the 254 hosts on the `154.30.109.0` network should be sent via the gateway `10.10.0.4` and use the `eth0` interface.

A reminder of the way it was written with the mighty `route` command is as follows. You can see that it's far from dissimilar to the `ifconfig` syntax you saw earlier:

```
# route add -net 154.30.109.0/24 netmask 255.255.255.0 gw 10.10.0.4
```

Hopefully, it's self-explanatory. The `gw` appendage obviously stands for "gateway". The only thing to note is the potential pitfall of the `-net` switch. The old `route` command was more concerned with being told about if you just wanted to redirect traffic to a new gateway for a single IP address or a whole network. The new kid on the block, however, figures that out with CIDR (Classless Inter-Domain Routing) formatting. That's the `/24` in this case: `154.30.109.0/24`.

The way that the old `route` command preferred to reference individual hosts was as follows with the `-host` adjustment (I've arbitrarily added `222` as the single IP address):

```
# route add -host 154.30.109.222 gw 10.10.0.4
```

However, a large number of Unix-like versions/distributions don't mind this shortened version too:

```
# route add 154.30.109.222 gw 10.10.0.4
```

Deleting a route is frighteningly simple. It's as easy as this command:

```
# ip route del 154.30.109.0/24 dev eth0
```

In the past if you wanted to alter the default route (clearly you know by now that I also mean the "gateway"), you would run the `route` command below. You have to do this carefully (!) and ideally use out-of-band access or SSHed in from another machine on the same LAN so that you didn't mind losing the default route.

For deleting a default route, compare the old and the new:

```
# route del default gw
# ip route del default
```

Adding a new default route is shown next; old and then new commands compared again:

```
# route add default gw 10.10.0.1 eth0
# ip route add default via 10.10.0.1 eth0
```

Clearly it's useful to mix and match these commands when using older operating systems. For example, I can easily commit `route del default gw` to memory it seems, but I frequently forget the `ip route del default` command.

Address Resolution Protocol

Another of my favorite protocols is the Address Resolution Protocol (ARP). It's simple, it's fast, and it is absolutely necessary to keep your machine on the network. The robust `ip` command does not shy away from ARP either.

As in the previous section, I'll provide old and new commands for comparison. Again with the `arp` command, I immediately use the `-n` switch without a second thought to avoid DNS load and latency.

Listing 11-4 shows which of the local hosts on the local area network (or within the VLAN—Virtual LAN) have recently communicated with the machine and registered an ARP entry as a result.

Listing 11-4. The Two Example Gateways Have Both Communicated with the Machine and Have Been Registered in the ARP Cache

```
# arp -n

Address       HWtype            HWaddress    Flags Mask      Iface
10.10.0.1      ether    00:a0:a3:e2:01:eb             C       eth0
10.10.0.4      ether    00:10:73:5a:26:12             C       eth0
```

One caveat is that I've heard some people (probably incorrectly) refer to the ARP "table," but I suspect it should really be referred to as the ARP *cache*. In essence it's a cache of machine IDs, involved in recent communications, within which each entry expires periodically, after a set expiration time. If you want to increase your ARP expiration knowledge about timeouts then there's a nice answer here: http://stackoverflow.com/questions/15372011/configuring-arp-age-timeout. The long and short of it is that don't just make a change to your kernel settings without understanding what you're doing first.

The equivalent syntax from the `ip` command is as follows and provides the output in Listing 11-5.

Listing 11-5. The Newer Type of Output Is Shown Using the ip n sh Command

```
# ip n sh

10.10.0.1 dev eth0 lladdr 00:a0:a3:e2:01:eb STALE
10.10.0.4 dev eth0 lladdr 00:10:73:5a:26:12 DELAY
```

Above you saw the shorthand, which is a more convenient way of expressing the following:

```
# ip neighbor show
```

Note the reference to `STALE` and `DELAY` in the newer style of output. A `STALE` neighbor is one that hasn't been spoken to for a while. The kernel probably assumes it's not available. A neighbor entry exhibiting a `DELAY` means that the kernel is dutifully trying to check in with the host but hasn't received any response. The `REACHABLE` host is, ahem, reachable, as you might imagine.

To add an entry to your ARP cache, you can simply use this command by associating the correct, hopefully unique (locally at least), MAC (Media Access Control) address with the IP address in question as follows:

```
# ip neigh add 10.10.3.2 lladdr 00:4d:20:15:b3:00 dev eth0
```

To delete an entry, use this:

```
# ip neigh del 10.10.3.2 dev eth0
```

Sometimes it is necessary to force an entry so it won't expire from the cache. You can do this with a simple appendage to the longer command. Apparently, the `nud` variable stands for "Neighbor Unreachability Detection".

The "permanent" switch has been covered and once added needs to be manually removed. The `noarp` option forces ARP to believe that this entry is kosher and it shouldn't attempt to check otherwise (it will just disappear as usual when its lifetime expires). The adjustable `reachable` option is as I've mentioned and the `stale` option means that ARP should treat an entry suspiciously, even though that ARP has recognized it as a valid entry.

An example `nud` addition to the previous command might be like this:

```
# ip neigh add 10.10.3.2 lladdr 00:4d:20:15:b3:00 dev eth0 nud perm
```

Now that you're fully versant in all things networking, I'm sure that you get the idea.

One final thing to look at with ARP is being able to flush your entire ARP cache. It's easy to do. There is some overhead, but if you're feeling brave you can empty the cache. You can flush the whole ARP cache using this `ip` command:

```
# ip -s -s neigh flush all
```

In case you think you're seeing double, the first `-s` adds greater levels of detail and stands for `-statistics`. The second points you at the "neighbor" table.

For comparison, in the olden days you might have cleared one IP address' entry as follows:

```
# arp -d 12.34.45.78
```

Banning Hosts

My favorite functionality available in Linux routing is being able to immediately ban a miscreant from getting access to my machine. (There are a few caveats of course; for one, you need the machine upstream to you, i.e., to your nearest router or gateway, to deflect the traffic so it gets nowhere near to your machine.) The following is still very useful in some circumstances such as minor Denial of Service (DoS) attacks.

The functionality I'm referring to has different names. The `ip` command's manual refers to adding a "blackhole". This is when traffic is not responded to but instead simply thrown into the "bit bucket," which is a bottomless pit where unwanted traffic is sent. Imagine it's like `/dev/null` on your filesystem. And, on that note, blackholing a route is called adding a "null route," because traffic is "nulled" or has nowhere to go. It's nipped in the bud and without wasting resources to respond to it with helpful errors in most cases.

An example of blocking such a nefarious route might look like this:

```
# ip route add blackhole 102.134.1.197/32
```

As you'd expect, you simply swap `add` with `del` to remove a route.

If you then run this command, you can check that it has been inserted into the local routing table. Its output is shown here:

```
# ip route show

default via 10.10.0.1 dev eth0 metric 100
blackhole 102.134.1.197
```

If you're interested, back in the day, you might have achieved similar results as follows:

```
# route add -host 102.134.1.197 reject
```

Summary

For some readers, this chapter may have just been a refresher but for those new to networking or to the `iproute2` package, hopefully it's a concise enough reference guide for the future.

With one package or another, sometimes a quick check on syntax that you've had positive results from in the past is necessary. Sysadmins have all sorts of ways to remember fiddly formatting and I tend to have an ever-growing text file that I should really add to a "Git repository" because it's cumbersome. To the newbies, please take heed and use these commands carefully. You should ideally test them on a development machine before wildly breaking things and irritating your employer or users.

It is important to be able to quickly recall the simple commands. I am forever using the shorthand `ip a` command, for example, which saves lots of typing along with the ability to append an IP address to an interface without having to look it up. As a result, when looking up help files or man pages, the remaining syntax is absorbed much quicker and causes less eyestrain. It is worth using the flexible `ip` command frequently until the syntax sinks in. It will almost definitely assist you in a crisis.

APPENDIX A

Hack, Break, Fix

While the security industry is constantly evolving its countering techniques, the categories of attacks suffered since the inception of the Internet have rarely changed. Both White Hats and Black Hats, whether testing with good intentions or nefariously attacking networks and systems respectively, generally only learn about a handful of different categories of attacks.

Understanding the many attack vectors that an attacker will target is of paramount importance for any White Hat. This appendix explores putting an older tool to good use on today's Internet. The fact that you can safely attempt exploits yourself and understand how to guard against them means that this invaluable tool rates very well in comparison to other security tools.

Step forward "Damn Vulnerable Linux" (known as DVL). If you want to learn about hacking, plugging security holes, and fixing broken systems, then to my mind the excellent DVL is still definitely worth looking at.

Exploring DVL

The DVL web site was taken offline around 2011 allegedly because it threatened to breach German security laws. Despite that, DVL was one of the most useful security technologies anyone interested in Linux could explore over the past few years.

You are actively discouraged from installing DVL onto a hard drive, so I will presume that you can configure your own VirtualBox (`https://www.virtualbox.org`), Virtual Machine (VM), or some other type of container. Alternatively, you can use DVL on its own live CD (without any permanent installation at all) or at a push you might have a suitably old machine whose drive can be happily wiped.

If you visit the SourceForge download URL mentioned there and then look at the list of ISO files, you can see a list of a few different DVL versions.

The link at the top saying "Looking for the latest version?" might suffice. There was a mention of the developer(s) working on version two of DVL and that link mentions `bt2final.iso`. If that's not working for you, you should have some success with downloading "DVL_1.5_Infectious_Disease.iso".

Once DVL is installed, you will see a screen mentioning the venerable BackTrack Linux (which is sadly now obsolete and its site now effectively points at Kali Linux; see `https://www.kali.org`), depending on your version. Following the onscreen instructions, you should then be able to log in using credentials such as these:

```
Login: root
Password: toor
```

Figure A-1 shows the basic GUI desktop manager of DVL version 1.5, despite a massive part of the work you will do on DVL being command-line based (in a shell).

Figure A-1. *An image showing the DVL desktop manager as found on the DistroWatch site; see* `http://distrowatch.com/images/cgfjoewdlbc/dvl.png`

By using older, more vulnerable, kernel versions, DVL can increase the attack surface of your VM considerably. You can definitely have a field day with some of the puzzles and exploitable software. Among the mountain of goodies, for example, are some nasty SQL (Structured Query Language) injections you can explore.

Lots of the system functions are purposely broken in varying degrees, such as the networking config. In terms of the packages, however, you can experiment with older versions of a web server (Apache), PHP, Database Server (MySQL) and an FTP server. You can also play with GCC (GNU Compiler Collection) and `strace`, among many other compiler packages, to your heart's content.

The former DVL web site made a big deal about not being able to cheat because there were no solutions provided. It's a nice claim and improves your bragging rights when you've achieved a hard-won level of success. Let's have a look at some of the applications bundled with DVL (without giving much away at all to avoid spoiling your fun).

Figure A-2 shows a sample of the bundled software that's designed to keep you on your toes.

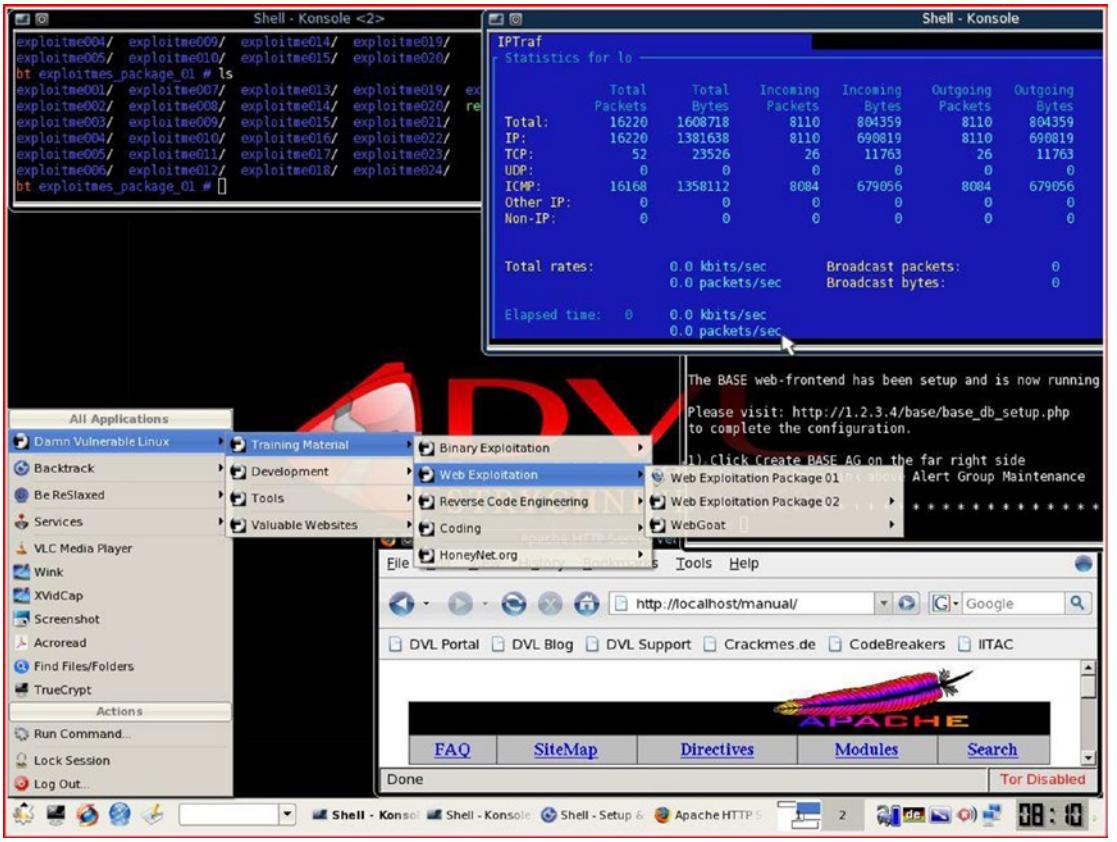


Figure A-2. *A snapshot from the DVL's Wikipedia page*

Depending on the version of DVL you are using, you should be confronted with a couple of web browsers (Firefox and Konqueror), some useful documentation (follow Damn Vulnerable Linux from the Start button), and a quick and easy access to a shell.

With a little digging, you can also look through a veritable treasure trove of utilities such as disassemblers and debuggers. If you've ever come across something called "breakpoint testing" or used watchpoints, these are exactly the tools that will assist you.

What's in the Box

As I've alluded to, the excellent testbed that is Damn Vulnerable Linux offers a number of different facets for you to hack, break, and fix. A number of very useful tutorials are also included. Previously, there were a number of videos on the DVL web site to complement these docs, but you might struggle find them online now.

Apparently around 65% of us are "visual learners," which means that we will respond best to learning from graphics and charts. There are a few users showing off their DVL skills and the odd sample can be found on YouTube it seems, but at a glance the original videos aren't available. The Wayback Machine can only offer so much after all (https://archive.org/web).

An excellent site called VulnHub (see https://www.vulnhub.com/series/damn-vulnerable-linux,1/) has a comprehensive list of that and is included in the DVL releases. For example, according to VulnHub in DVL version 1.4 (codenamed "Strychnine"), you can expect to find the following contents:

- [Application Development] Add Motor IDE
- [Application Development] Update HLA to 1.98 and StdLib to 2.3
- [Application Development] Add LogWatch
- [DVL Core] Add XEN
- [Reverse Code Engineering] Add Insight GDB Debugger
- [Tutorials] Add CPU Sim - An Interactive Java-based CPU Simulator
- [Reverse Code Engineering] Add JAD Java Decompiler
- [Tools] Add VLC Media Player
- [Documentation] Add TeTex
- [Documentation] Add JabRef
- [Application Development] Add Kile
- [Documentation] Add kDissert Mindmapper
- [Peneration Testing] Add JBroFuzz
- [Application Development] Add WebScarab
- [Peneration Testing] Add CAL9000
- [Reverse Code Engineering] Add KDBG
- [Application Development] Add xchm
- [DVL Core] Add gtk libs
- [Tools] Add xvidcap
- [Tools] Add AcroRead
- [Tools] Add Scite

For a teaser (I promised not to discuss details for good reason), the following list shows what was included with DVL v1. This release was apparently only a collection of tools from which the VM could be further developed on as opposed to hacking, breaking, and fixing VM.

- HT 0.5
- libreadline4_4.2a-5_i386
- gdb_5.2.cvs20020401-6_i386
- binutils_2.12.90.0.1-4_i386 (including objdumps,gas,strings ...)
- nasm-0.98-1.i386

- HLA v1.86
- libelfsh0-dev_0.65rc1-1_i386
- elfsh_0.65rc1-1_i386
- Apache 2.0.5.4
- Php 4.4.0
- ethereal-common_0.9.4-1woody12_i386
- ethereal_0.9.4-1woody12_i386
- libpcap0_0.6.2-2_i386
- tcpdump_3.6.2-2.8_i386
- lsof_4.57-1_i386
- ltrace_0.3.26_i386
- nmap_2.54.31.BETA-1_i386
- strace_4.4-1.2_i386
- ELFkickers-2.0a (including sstrip, rebind, elfls, ebfc, elftoc)
- GCC/G++ 3.3.4
- GNU Make 3.80
- bastard_bin- 0.17.tgz
- Mysql-server 4.4.1
- Ruby 1.8
- Python 2.3
- lida-03.00.00
- DDD 3.3.1

Vulnerable

It is difficult to ignore the VulnHub site. If you're ever lost for a hobby, you could do worse than lose a weekend or two using some of the VM images on VulnHub. It's a collection of pre-built images, among other things, that you are encouraged to exploit. Here's an example of a stated goal as found on the "Acid: Reloaded" VM: "Escalate the privileges to root user and capture the flag".

You might also want a look at this demo if SQL exploits interest you. It's called "codebashing" (`http://www.codebashing.com/sql_demo`) and the site talks about learning about securing your applications. There are paid-for options and the site promises that programmers will teach you all about the ever-important application security.

A community project that aims to allow users to upload sandbox applications for others to hack is aptly named `https://hack.me`. What I like about this site is the diverse variety of what it calls "hackmes". If you visit `https://hack.me/explore` then there's web forms, PHP bugs, and other varieties available. There's a "hackme" called "Delete All The Things," which kindly requests that you use SQL injections to delete the entire `users` table in the database. The nicely built site is certainly worth a look and adding a contribution to if you're feeling inclined.

It seems that Java is all the rage at the moment. There's a purposely-insecure J2EE (Java 2 Platform Enterprise Edition) application that aims to help teach security concepts called WebGoat (https://www.owasp.org/index.php/WebGoat_User_Guide_Table_of_Contents). Once you have Tomcat up and running, you are asked to do the following with each security assessment:

- Examine client side content, such as the HTML and scripts.
- Analyze communications between the client and server.
- Review any cookies and other local data.

You simply point your local browser at http://127.0.0.1/WebGoat/attack to get started. You can use these credentials to gain access:

```
Login: guest
Password: guest
```

Along the same vein, you could probably get lost in WebGoat for a number of days.

Finally, here's a piece from back in the day (written in 1999), which looks at a Buffer Overflow that exploits a program with SUID permissions. It's well written and provides insight into what goes on behind the scenes during an attack (or at least what used to go on back in the year 1999); see http://www.linuxjournal.com/article/2902.

Summary

You could probably soak up a number of days attempting to exploit, harden and repair some of the code and VMs that we've looked at. There are literally thousands of security tools available to both attackers and those who want to protect their systems.

If you haven't seen the likes of these tools before, it's improbable that you will be immediately able to step into a career in penetration testing or become a security professional. You should, however, be armed with a number of very useful tools that help to get you part of the way there. If you are interested in this area, you could do much worse than have a look at the excellent, comprehensive package called Metasploit (http://www.metasploit.com). It provides a framework that helps inject exploits into remote machines in order to essentially monitoring how they react. You work with payloads and choose which bug to take advantage of, among other aspects. There's also an eye-opener of how to encode your attack traffic so that firewalling (intrusion detection systems) ignore the payload as it is presented to their filtering rules. All in, it's potentially a very complex tool but a fantastic learning experience that will take you one step further than many other available tools.

Hopefully, your arsenal now includes enough extra security weaponry to help you do more than the bare minimum to keep your services running.