Post Installation

COSC301 Laboratory Manual

In the previous lab, we installed our server virtual machine, Server1, and applied a bunch of updates. In this lab, we're going to continue from there, and actually make it part of the same network as our client, set up some IPv4 and IPv6 addressing and run a few tests. We shall also spend some time looking at one way of creating network services, as well as one common way to restrict access to network services.

> **Important**
>
> Except where explicitly noted (typically in command prompts, but also in the text), all work is to be performed on the server you installed previously. You will also need Client1, but nothing else.
>
> Also starting from this lab, you will see the # prompt for commands, which means the commands need root privilege. In such cases, you should use **sudo** to execute the commands, instead of executing them as root user. This should be always the practice unless specified otherwise explicitly as in some advanced labs where you do need to login as root user to perform certain tasks.

1. Adding the "inside" Interface

Currently, the "outside" interface of Server1 is configured to attach to VirtualBox's "NAT", allowing it to access the outer (campus) network without the server needing an address on the outer network. But with this attachment comes a restriction; we can't talk to other virtual machines. We want our virtual machines to all be connected to each other in a dedicated LAN.

We could re-attach the "outside" interface of Server1 to the internal network, which would allow us to connect with Client1, but that means we wouldn't be able to connect to the outer network if we needed packages. That's really annoying.

To solve this problem, we shall add another adaptor to VirtualBox, and configure the new interface as the "inside" interface. This will allow us to create a "dual-homed" configuration whereby we can talk to both the outside world and the internal network.

Shutdown the server cleanly. It needs to show its status as "Powered off". If it is showing as "Saved" you will need to start it, then shut it down cleanly using **shutdown -h now**.

In the VirtualBox network setting for the server, leave Adaptor 1 as it is (it should be attached to NAT), and go into Adaptor 2. Enable the interface and attach it to the Internal Network "COSC301 Internal Network 1".

Start the server. To prevent confusion, use the same method we used in earlier labs to add then change the interface name from "enp0s8" to "inside" (hint: `/etc/systemd/network/70-intnet.link`, and **update-initramfs -u**).

Change the network interface configuration inside the server by editing the file `/etc/network/interfaces`.

```
auto lo
iface lo inet loopback
```

```
auto outside
iface outside inet dhcp

auto inside
iface inside inet static
    address 192.168.1.1
    netmask 255.255.255.0
```

Affect the changes by rebooting.

```
$ sudo shutdown -r now
```

Exercise

As an exercise check that everything is now as expected. Are the interfaces configured as we would expect?

```
$ ifconfig inside
…IP address should be 192.168.1.1
$ ifconfig outside
…IP address should be 10.0.2.15
```

Okay, now check that the routing table is as we expect, with a default route going out the "outside" interface:

```
$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         _gateway        0.0.0.0         UG    0      0        0 outside
10.0.2.0        0.0.0.0         255.255.255.0   U     0      0        0 outside
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 inside
```

To find out the IP address of _gateway:

```
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.2.2        0.0.0.0         UG    0      0        0 outside
10.0.2.0        0.0.0.0         255.255.255.0   U     0      0        0 outside
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 inside
```

Finally, a "test by doing": can we get to the package repository?

```
# apt-get update
…
Fetched 565kB in 6s (87.56kB/s)     Success!
Reading package lists... Done
```

Testing by doing

How might you test if a web server is working? You could look to see if it running (**pstree** etc.), but that isn't an exhaustive test. It is much better to simply retrieve a page (possibly multiple pages), which tests the entire "stack". Testing by doing is a great way of testing for success conditions, but is not a good way of diagnosing faults.

Okay, so now we know that Server1 can talk to the world. Let's re-introduce Client1 into the network so we can test that we can communicate with hosts in the internal network.

Important

Start Client1, run the command **sudo apt install resolvconf** in a terminal window to install the resolvconf package. Then shutdown Client1

Before you start Client1 again, go into its settings and ensure that the network adaptor Adaptor 1 is connected to the Internal Network "COSC301 Internal Network 1". Then start Client1.

If you recall, in the lab on basic interface management, we only gave Client1 a temporary IP address, and did not make a permanent configuration. Let's create one now. **On Client1**, edit `/etc/network/interfaces` and *add* the following stanza for `iface eth0` or *replace* the stanza if there is one:

```
auto eth0
iface eth0 inet static
    address 192.168.1.11
    netmask 255.255.255.0
```

Affect the change using **ifup**:

```
client1# ifup eth0
 … should have no output if it works
client1$ ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.11  netmask 255.255.255.0  broadcast 192.168.1.255
        ether 08:00:27:f8:ef:dc  txqueuelen 1000  (Ethernet)
        RX packets 151  bytes 10752 (10.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 14  bytes 1076 (1.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Exercise

As an exercise ensure you get the IPv4 address specified. Make sure that you have successfully given Client1 its address.

We added Client1 to the network to ensure that Server1 could talk to other internal hosts, so let's test that now. Still on Client1, let's check that we can ping Server1:

```
client1$ ping -c2 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=3.39 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=0.358 ms

--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.358/1.874/3.391/1.517 ms
```

That worked, which means traffic can flow in both directions. To be doubly sure[1], let's try the same thing on Server1:

```
server1$ ping -c2 192.168.1.11
PING 192.168.1.11 (192.168.1.11) 56(84) bytes of data.
64 bytes from 192.168.1.11: icmp_seq=1 ttl=64 time=4.88 ms
64 bytes from 192.168.1.11: icmp_seq=2 ttl=64 time=0.385 ms
```

[1] At present, its not actually needed, but in general, because of devices such as firewalls and NAT, it pays to test both directions if needed.

```
--- 192.168.1.11 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.385/2.636/4.887/2.251 ms
```

Hooray! So we know local deliveries are working on the internal network. Time to go onto the next section.

2. Configuring Basic NAT

So local deliveries work, but soon we shall want to access the outer network from Client1, so let's test remote deliveries using the same "test by doing" as we did earlier on the server:

```
client1# apt-get update
…
W: Failed to fetch http://…  … Temporary failure resolving …)
E: Some index files failed to download, …
```

What happened there? The message "Temporary failure resolving …" indicates that it failed to convert an Internet name into an Internet address. This is a component called "DNS", which we shall see later.

Because we will be encountering DNS later on, we shall just skim over the necessary configuration. First, let's just ensure that Client1 is configured to access a DNS correctly. Add in the file `/etc/resolvconf/resolv.conf.d/head` on Client1 the following lines:

```
nameserver 139.80.64.1
nameserver 139.80.64.3
```

Once you have these lines, run **$ sudo resolvconf -u** or **$ sudo service resolvconf restart** to make the change effective.

These are the same DNS servers that your lab iMac is configured to access, and will be different on different networks.

So how do we get to those 139.80.64.* addresses, which are not on the local network; do we have a route which would allow Client1 to get there?

```
client1$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 eth0  Ignore
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
```

There is no route, such as a default route, to allow us to get out of the network. Open up `/etc/network/interfaces` on Client1, and edit the "eth0" stanza to add a default route via a gateway:

```
auto eth0
iface eth0 inet static
    address 192.168.1.11
    netmask 255.255.255.0
    gateway 192.168.1.1
```

Bring the interface down then up again:

```
client1# ifdown eth0
SIOCDELRT: No such process  Trying to delete the gateway, which has not yet been added
client# ifup eth0
```

```
client$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    100    0        0 eth0   Success!
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 eth0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
```

Okay, so now can we get out to the wider network?

```
client1# apt-get update
0% [Working]  Long hang
^C
```

It took a long time because there is an error. Rather than show you how you could diagnose it (we can save that for a later lab), we'll simply tell you that it's a routing issue: packets aren't being forwarded by Server1, because it's not configured to do so. On Server1, enable IP forwarding by setting the appropriate system control ("sysctl"), which configures kernel behaviour. We can do this persistently by editing the file `/etc/sysctl.conf` on Server1:

```
…
net.ipv4.ip_forward = 1   Uncomment this line
…
```

```
server1# sysctl -p
net.ipv4.ip_forward = 1   Prints changes it applies
```

However, this is not the end of the problem. Although packets can now be forwarded from Server1's inside to outside, VirtualBox's "NAT" attachment doesn't have a "return route" (it doesn't know that in order to get packets to 192.168.1.0/24 it should go through 10.0.2.15 (Server1's outside interface). Because we can't add a return route (VirtualBox doesn't expose such functionality, we shall have to add another level of NAT — it's ugly, but it will work.

Confused? Don't panic!

At this point, you don't need to really understand what we're doing right now, it's just necessary stuff we need to do in order to get a network that we can further play with. We'll meet it again in more detail later, at which point you should come to understand.

To enable NAT, write the following onto `/etc/network/if-up.d/nat` on Server1:

```
#!/bin/sh

if [ "$MODE" = start -a "$LOGICAL" = outside ]; then
    echo "Enabling NAT on 'outside' interface"

    iptables -t nat -F
    iptables -t nat -X
    iptables -F
    iptables -X

    iptables -t nat -A POSTROUTING -o outside -j SNAT --to-source 10.0.2.15
fi
```

Use **chmod** to make the script executable. This script should now run when we use **ifup**, after the interface has been bought up. To test, first take the interface down, then bring it back up.

```
# ifdown outside
…
```

```
# ifup outside
… messages from DHCP client
Enabling NAT on outside   success
```

Does it work yet? Now try again on Client1:

```
Client1# apt-get update
… should work to completion
```

> **Exercise**
> Hooray it works! As an exercise take a little break. In the next section, we'll enable IPv6 connectivity on the local link. Don't worry, it'll be pretty easy, we won't even have to do anything on Client1!

3. Configuring IPv6 Router Advertisements and a Static Address

At this point, we have Client1 and Server1 reachable via IPv4, and Client1 can even connect to addresses beyond Server1, also over IPv4. In this section, we're going to show you how to set up simple IPv6 connectivity, but only for the local link (because going beyond Server1 using IPv6 is not yet supported on our outer network).

Client devices generally require no configuration. They will by default act on router advertisements. We don't have a router on our internal network that sends out router advertisements, so we shall add that feature to Server1 as well, as it is going to be a router (albeit only for IPv4 at present).

When we've done that, we shall configure Server1's inside interface with a static IPv6 address. This is important because servers generally need static addresses, and because Server1 is acting as a router, it will not participate in the SLAAC (StateLess Address AutoConfiguration) process, which was introduced in the lab of IPv6 Bootcamp.

> **Note**
>
> In the IPv6 Bootcamp lab, we used a virtual appliance called Radv to send out router advertisements. That appliance is now made redundant by Server1 and should not be enabled on your internal network.

The first thing we have to do is to make Server1 perform the functions of SLAAC, in order to send out router advertisements from Server1 (which is our router). The software for doing this on Linux systems is generally the **radvd** package, which we could install using the Debian/Ubuntu package of the same name

```
# apt-get install radvd
…
Setting up radvd  …
 …
 …
```

At this stage, it won't be able to start because there is no configuration file. This is good because no default configuration file means we are not going to risk polluting the network with poor router advertisements. We just need to simply create the configuration file `/etc/radvd.conf` as root:

```
interface inside
{
    AdvSendAdvert             on;

    prefix fd6b:4104:35ce::/64
    {
        AdvOnLink             on;
        AdvAutonomous         on;
    };
};

interface outside
{
    AdvSendAdvert             off;
};
```

Check radvd.conf(5) for more information about this file format. This is a fairly minimal example, and makes use of radvd's default values for most things, which are sensible defaults.

According to the `README.Debian` file (mentioned in the startup attempt of **radvd**), IPv6 forwarding needs to be enabled (even though at present we're not going to be doing any forwarding of IPv6). So uncomment or add the following line to `/etc/sysctl.conf`:

```
net.ipv6.conf.all.forwarding=1
```

Confirm the changes of sysctl, then start **radvd**:

```
# sysctl -p
net.ipv6.conf.all.forwarding = 1  prints changes it makes
# /etc/init.d/radvd start
Starting radvd: radvd.
```

Check that it's up and running:

```
$ ps -eo pid,command | grep radvd
 2057 /usr/sbin/radvd …
 2058 /usr/sbin/radvd …
 2141 grep --color=auto radvd    ignore this line
```

You may find that it's not running, in which case check the status of the service.

```
# service radvd status
● radvd.service - LSB: Router Advertising Daemon
   Loaded: loaded (/etc/init.d/radvd; bad; vendor preset: enabled)
   Active: active (exited) since Fri 2018-04-13 02:14:38 UTC; 2min 41s ago
     Docs: man:systemd-sysv-generator(8)

Apr 13 02:14:38 ubuntu systemd[1]: Starting LSB: Router Advertising Daemon...
Apr 13 02:14:38 ubuntu radvd[5333]: Starting radvd:
Apr 13 02:14:38 ubuntu radvd[5333]: * /etc/radvd.conf does not exist or is empty.
Apr 13 02:14:38 ubuntu radvd[5333]: * See /usr/share/doc/radvd/README.Debian
Apr 13 02:14:38 ubuntu radvd[5333]: * radvd will *not* be started.
Apr 13 02:14:38 ubuntu systemd[1]: Started LSB: Router Advertising Daemon.
```

You'll see that the service is active (exited). This means that it tried to startup, but failed to so and has stopped. It stopped because it couldn't find the /etc/radvd.conf file when it started. You will need to restart the service (as opposed to just starting it). There are two ways to do this, firstly, by stopping then starting it, or simply issueing restart. In addition, enable the *radvd* service so that it can be automatically started after reboot.

```
# service radvd restart
```

```
# systemctl enable radvd
```

Double check the status as before.

```
# service radvd status
● radvd.service - LSB: Router Advertising Daemon
   Loaded: loaded (/etc/init.d/radvd; bad; vendor preset: enabled)
   Active: active (running) since Fri 2018-04-13 02:23:26 UTC; 9s ago
     Docs: man:systemd-sysv-generator(8)
  Process: 5729 ExecStop=/etc/init.d/radvd stop (code=exited, status=0/SUCCESS)
  Process: 5737 ExecStart=/etc/init.d/radvd start (code=exited, status=0/SUCCESS)
   CGroup: /system.slice/radvd.service
           ├─5746 /usr/sbin/radvd -u radvd -p /var/run/radvd/radvd.pid
           └─5747 /usr/sbin/radvd -u radvd -p /var/run/radvd/radvd.pid

Apr 13 02:23:26 ubuntu systemd[1]: Starting LSB: Router Advertising Daemon...
Apr 13 02:23:26 ubuntu radvd[5745]: version 2.11 started
Apr 13 02:23:26 ubuntu radvd[5737]: Starting radvd: radvd.
Apr 13 02:23:26 ubuntu systemd[1]: Started LSB: Router Advertising Daemon.
```

That looks healthier, shall we have a look at what it's doing on the network?

```
# lsof -Pni
nothing related to radvd!
```

Apparently it doesn't use IPv6 sockets at all: can't believe that, let's have a closer look:

```
# lsof | grep radvd
…lot's of lines, including a couple like this one:
radvd … raw6 …
```

Ah, so **radvd** does its work by using "raw" IPv6 sockets. It does this so it can specify exactly what to put in the header fields, even if the operating system doesn't have library support for newer IPv6 header options.

Anyway, let's get back to our testing. Start up Client1 connected to the same internal network. When its interface comes up, it should send out a Router Solicitation, which should cause **radvd** to send out a Router Advertisement. Let's see what addresses Client1 has generated for itself:

```
Client1$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 08:00:27:99:c2:7d
          inet addr:192.168.1.11  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fd6b:4104:35ce:0:a00:27ff:fe99:c27d/64 Scope:Global
          inet6 addr: fe80::a00:27ff:fe99:c27d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:624 errors:0 dropped:0 overruns:0 frame:0
          TX packets:250 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:820367 (820.3 KB)  TX bytes:29767 (29.7 KB)
```

Exercise

As an exercise make sure the interface has a Unique-Local Address. Did you get a default route? Use the following command to make sure Client1 has got an address and a useful default route. Mind you that we haven't had to do any configuration on Client1 to get that.

```
Client1$ ip -6 route
fd6b::4104:35ce/64 dev eth0 …
```

```
fe80::/64 dev eth0 …
default via fe80::a00:27ff:fe69:e1ae dev eth0 …  success
```

You might perhaps be thinking it odd that we did in fact get a default route. After all, you would think that you should have to manually configure Client1 to get a default route, but this is not the case. With IPv6, it makes it easier to have multiple routers that a host can use for a default route, and they each advertise a "default router priority", such as low, medium or high, which allows for graceful failover.

So now that Client1 has a fd6b:... address (a "Unique-Local Address", or ULA for short), we should be able to make connections using it. The only other thing in our network in the moment is Server1. Does it have a ULA?

```
$ ifconfig inside
inside    Link encap:Ethernet  HWaddr 08:00:27:50:e0:93
          inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe50:e093/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:297 errors:0 dropped:0 overruns:0 frame:0
          TX packets:701 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:31067 (31.0 KB)  TX bytes:828807 (828.8 KB)
```

No, it doesn't have a ULA, only a LLA (Link-Local Address). Why is that? It is because Server1 is configured as a router, no longer a host. As such, it does not generate SLAAC addresses. So we need to configure a static address, which is really very easy. On Server1, edit `/etc/network/interfaces`, and make the alterations as indicated:

```
…
auto inside
iface inside inet static
    address 192.168.1.1
    netmask 255.255.255.0
iface inside inet6 static
    address fd6b:4104:35ce::1
    netmask 64
```

Affect the change using **ifup**:

```
# ifdown inside
…
# ifup inside
…
$ ifconfig inside
inside    Link encap:Ethernet  HWaddr 08:00:27:50:e0:93
          inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fd6b:4104:35ce::1/64 Scope:Global      Success
          inet6 addr: fe80::a00:27ff:fe50:e093/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:297 errors:0 dropped:0 overruns:0 frame:0
          TX packets:709 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:31067 (31.0 KB)  TX bytes:829623 (829.6 KB)
```

Great, so now both Client1 and Server1 have a ULA. Let's just check that we can communicate using them with **ping6**; because this is a ULA, and not a LLA, we do not need to supply a scope identifier (eg. `%inside` or `%eth0`, as we have done previously).

```
Client1$ ping6 -c1 fd6b:4104:35ce::1
PING fd6b:4104:35ce::1(fd6b:4104:35ce::1) 56 data bytes
64 bytes from fd6b:4104:35ce::1: icmp_seq=1 ttl=64 time=0.669 ms
```

```
--- fd6b:4104:35ce::1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.669/0.669/0.669/0.000 ms
```

Exercise

So now we have Client1 and our server with both IPv4 and IPv6 addresses. As an exercise let's just do some brief testing.

```
Client1$ ping6 -c1 fdb6:4104:35ce::1
… Destination unreachable: No route
Blast! Typing mistake.
Client1$ ping6 -c1 fd6b:4104:35ce::1
… 64 bytes from fd6b:4104:35ce::1 … success!
Client1$ ping -c1 192.168.1.1
64 bytes from 192.168.1.1 … success!
```

These IPv6 addresses are rather more cumbersome to type and recognise (but thankfully after a while, the shorter static addresses, are reasonably easy) . Let's make it easier on ourselves: **on both Client1 and Server1**, *add* the following into `/etc/hosts`, remembering to use the correct address for Client1, which will be different from that shown below:

```
fd6b:4104:35ce::1 ip6-server1
fd6b:4104:35ce::a00:27ff:fe99:c27d ip6-client1
```

What this does is to make a mapping between an address and a name. It is important to realise that these changes are only visible to these two machines: if you were to have more machines they would not see the change. We shall improve on this situation when we encounter DNS.

```
Client1$ ping6 ip6-server1
64 bytes from ip6-server1 … much easier!
```

```
Server1$ ping6 ip6-client1
64 bytes from ip6-client1 …
```

What we've done is to create a simple little shorthand. You may be wondering why we bothered to put the (informal) prefix of "ip6-". That is simply for ease of debugging, making it easier to recognise/specify the use of IPv6 or IPv4. At this stage, you don't need to do anything similar for IPv4. This is just a little convenience for ourselves to make it easier to deal with IPv6.

4. Pruning Services

Most operating systems seem to come loaded with services enabled out of the box; most of which you don't need and therefore ought not to be running. Ubuntu, on the other hand, has a good policy of not shipping with any network services reachable from outside the machine by default, though naturally if you ask for a particular network service to be installed, it will be reachable.

In order to give us something interesting to look at in this section, you will first need to run the following "mystery commands" on Server1, which will install and configure some services for us to look at:

```
# apt-get install openbsd-inetd netcat-openbsd openssh-server
# sed -i -e 's/^#daytime/daytime/' /etc/inetd.conf
# /etc/init.d/openbsd-inetd restart
```

What that does, in short, is to install some software, which includes an "Internet Super-Server"[2], which we shall be using shortly; alter its configuration file a little to enable a particular service by uncommenting a line; and restart it, affecting the change. We also install the OpenBSD version of the useful netcat utility (the "TCP Swiss Army Knife"), as it again has much better IPv6 support than other versions of netcat[3]. Finally, we also installed the OpenSSH SSH service. We can now run **lsof** and see what is listening on the network:

```
# lsof -i
COMMAND    PID USER   FD TYPE … NODE NAME
dhclient  294 root   4u IPv4 … UDP *:68
sshd       902 root   3u IPv4 … TCP *:ssh (LISTEN)
sshd       902 root   4u IPv6 … TCP *:ssh (LISTEN)
inetd     1275 root   4u IPv6 … TCP *:daytime (LISTEN)
```

Note

If you don't see any lines output, check that you have run **lsof** with root privilege. Otherwise, it will only report on your own processes, of which you will most likely have none that are using network sockets.

`dhclient` is the DHCP client for the NAT network attachment (we'll learn more about DHCP in a later lab). Likewise, we'll look at that `sshd` entry a little later, but right now we want to have a look at the `daytime` entry, and find out more about it.

You'll notice that **inetd** is handling the connections for the `daytime` service. The configuration file for **inetd** is `/etc/inetd.conf`. Before we disable anything in `/etc/inetd.conf`, we should know that a service is represented as a pair of **`protocol`/`port`**, where `protocol` is either `tcp` or `udp`, and `port` is the port number. This is a common way of specifying port numbers of services in documentation. To find the protocol and port number of a service, you can use the `-Pni` options to **lsof**. You can google **`protocol`/`port`** to find out information about the service.

It is useful also to look at what interfaces services are listening on, for that tells us much about where a service is being offered. For the `daytime` entry above, a * indicates it is listening on all IPv6 interfaces (furthermore, in a common dual-stack feature, because there is no IPv4 service on the same port number, IPv6 will also likely get IPv4 requests as well). If you see a particular hostname (or IP address if you are using the `-n` option to **lsof**), then it is only accepting connections on that interface. Commonly, services that need to be running, but don't need to be remotely accessible by default will listen on 127.0.0.1 or ::1. Such an address is commonly shown as `localhost`, `ip6-localhost` or the *canonical hostname* of the machine, depending on the contents of `/etc/hosts`, which differs on different distributions.

inetd is configured via the file `/etc/inetd.conf`. An entry in that file is disabled either by commenting it out or removing it entirely. To make it easier for package installation scripts to be run, various configuration files can be managed using system-provided scripts, such as **update-inetd**[4]. This is easier than having to have each inetd-related package having to edit the file itself, separating policy and implementation. In this lab, we disable the "daytime" service manually by editing `inetd.conf`.

Once you have modified `inetd.conf`, you need to tell **inetd** that it has changed. This is generally done by the services startup script, such as by using the command **/etc/init.d/openbsd-inetd reload**.

[2]This particular version comes from OpenBSD, and supports IPv6 much better than the other available version: Inetutils-inetd.
[3]As an example, the netcat6 package doesn't allow you to specify an IPv6 address to connect to, only a hostname.
[4]This would replace the call to **sed** we used earlier, and would be expected to be more robust.

Note

In summary, to disable or enable a service from **inetd**, you generally want to use a pattern of commands such as the following:

1. **vim**
   Use **vim** or **nano** to edit `inetd.conf` and comment or uncomment the corresponding line of the service (judging by **`protocol`**/**`port`** of the service).

2. **pstree**
   Use this to verify that the right number of daemon processes is running. In the case of **inetd**, there should only be one **inetd** process in a quiescent system (meaning the system is not serving any requests.)

3. **sudo /etc/init.d/openbsd-inetd stop**
   Use this to shut down the service.

4. **sudo killall inetd**
   You can use this to kill off any remaining **inetd** processes so that processes get a second chance to terminate gracefully. Give a little time for them to be shut down first though.

5. **pstree**
   Has it gone yet? If not, use **sudo killall -KILL inetd** to kill it forcefully. Check that it has really died this time with another **pstree**.

6. **sudo /etc/init.d/openbsd-inetd start**
   This will start the service again. Note that if you have commented out all services in `inetd.conf`, **inetd** will fail to start, which is ok.

7. **pstree** and **lsof -ni**, to verify that the process is running, and listening as expected.

8. Check your system logs, typically in `/var/log/syslog`. You can use the **tail** command to view the end of this, but beware that viewing logs using **tail** exposes a security weakness[5]

Following this general procedure throughout the rest of the course will greatly help you in diagnosing problems, and save yourself a lot of time in the future.

Now that we've disabled the "daytime" service, let's just ensure that it is no longer available. Here, We're using a particular invocation of **lsof** that just tells us about things on the "daytime" port:

```
# lsof -Pni:daytime
```

Because there is no output reported (and yes, we are running this with root privilege), we can infer that nothing is listening on the "daytime" port.

[5]Discussed in Hacking Linux Exposed [http://www.hackinglinuxexposed.com/]. You should instead create an alias in your `~/.bashrc`, such as `alias vlog='sudo less --follow-name +G +F'`.

What about that SSH service?

Here is what is currently listening:

```
# lsof -Pni
COMMAND    PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
dhclient …
sshd      9723 root    3u  IPv4  23591      0t0  TCP *:22 (LISTEN)
sshd      9723 root    4u  IPv6  23593      0t0  TCP *:22 (LISTEN)
```

Let's assume that we don't need the SSH service; how best to disable it? Unlike the "daytime" service, which was run by **inetd**, the SSH service is run by the **sshd** process. We have a number of options available to disable the service, listed in no particular order of suitability:

1. Remove the links from under `/etc/rc*.d/Snnname` that cause the associated script in `/etc/init.d/name` to be run.

   This is generally the best way to disable a background service either permanently or have it not running by default. You can stop a service with **/etc/init.d/name stop**. It can still be run by hand by calling **/etc/init.d/name start** etc.

   There is generally a tool to help you manage the contents of `/etc/rc*.d/`. On Debian-based systems, you can use the **update-rc.d** script[6].

2. Uninstall the package, retaining its configuration files. This can be done using **apt-get autoremove packagename**. This is useful if you think you may want to keep the configuration files and data files for a later time. It also means the program cannot be (accidentally or otherwise) run. Note that we have used `autoremove` instead of `remove`, which removes any other packages that were installed to satisfy a dependency that are no-longer needed.

   This is generally the best way to remove a service if it doesn't need to be run anymore. One particular risk is that the package could accidentally be installed at a later date if it gets installed due to installation of other packages via package dependency.

3. Uninstall the package, purging its configuration files. This can be done using **apt-get autoremove --purge packagename**. This is useful when you want to permanently remove a package, or you don't want stale (possibly broken) configuration files hanging around, which might confuse issues later if you decide to re-install the service. Your original configuration files should be in backup and ideally in version control before the uninstallation.

In our case, we don't want the SSH service installed at all, so we shall practice removing the package (in a later lab, we shall install it when we need it).

To make things a little more real for a beginning administrator, let's assume that we don't know what package contains this `sshd` process. Where is this **sshd** process anyway?

```
$ ps -eo command | grep sshd
/usr/sbin/sshd
…
```

Okay, so it seems that the `sshd` mentioned in the **lsof** output above is actually `/usr/sbin/sshd`. What package contains that file?

[6] On Redhat systems, the equivalent is **chkconfig**.

```
$ dpkg -S /usr/sbin/sshd
openssh-server: /usr/sbin/sshd
```

Now we know the package name, we should be able to remove it:

```
# apt-get autoremove openssh-server
…
```

Verifying that it is no-longer running:

```
# lsof -Pni:22
No output, therefore nothing listening on port 22 (ssh)
```

5. The Internet Super Server

In this section we shall create a new **inetd** service, which illustrates how easily creating an service can be when using an Internet Super Server. In the next section, we shall go on to restrict access to it using TCP-Wrappers.

Procedure 1. Tiny File Server

1. As root (ie. using **sudo**), create the file `/usr/local/sbin/tinyfs` and mark it executable. This file is a simple Perl script; the contents are listed below (To save the editing time, you could skip the comments):

```
#!/usr/bin/perl -w
#
# Naive file server. Gets a filename in the first line from stdin,
# removes any CRLF line-ending, and sends out the file that has been
# requested. Provides NO authentication, NO authorisation, NO extra
# access control, and NO accounting; this should be run as the 'nobody'
# user and guarded with at least TCP Wrappers. Best not to use this
# in production, it is here to illustrate the basic principles of inetd.
#
# Lines are terminated by CRLF, the internet standard line-ending.

use strict;

$/ = '\r\n';            # read records (lines) terminated by CRLF
my $filename = <>;      # read a record (line) from stdin
chomp $filename;        # remove line ending

# Open the file, or die trying. Any error would be sent to stderr, which
# is connected to the client also, so typically you don't want to send
# anything to stderr!
#
# Seeing as this would have to implement some sort of application-layer
# network protocol, we should have some sort of result code. We'll just
# either say "OK\r\n" or "ERROR: reason\r\n" on the first line of the
# result.

if (open FILE, '<', $filename) {
    print "OK\r\n";
} else {
    print "ERROR: $!\r\n";
    exit
}

# Note that in this case, the files are output verbatim, no line-ending
# translation is performed (like FTP 'BINARY', not like FTP 'TEXT')

$/ = 4096;              # read 4kB of data at a time
```

```
while(<FILE>) {         # while we can read a record...
    print;              # ...print it
}
close FILE;
```

This will offer a very naïve and rather insecure file transfer service. You will notice that it reads from stdin, and writes to stdout. This is the basic principle of any server that uses the Internet Super Server. The stdin, stdout and stderr(!) get connected to the TCP (or UDP) socket, and so communicates with the remote peer.

Before we attempt to use it over the network, let's just demonstrate how it works, showing that it doesn't have to know anything about networking.

```
$ echo '/etc/hostname\r\n' | tinyfs
OK          our result code
server1     /etc/hostname is very short, only one line
```

2. Add the following entry to `/etc/inetd.conf`.

```
tinyfs stream tcp4 nowait nobody /usr/local/sbin/tinyfs tinyfs
tinyfs stream tcp6 nowait nobody /usr/local/sbin/tinyfs tinyfs
```

What does this mean? The `tinyfs` will be its service name: it will tell **inetd** which port to listen on. The `stream` specifies that it is a stream protocol (and not a datagram protocol); the `tcp4` and `tcp6` says we are using TCP (which is always a stream protocol) over IPv4 or IPv6 respectively. `nowait` tells **inetd** it may process multiple connections at once, rather than having to wait for one to finish before dealing with another. `nobody` is the user the service should run as. If this were `root`, then sensitive files, such as `/etc/shadow`, which are normally only readable by the root user, would be world readable, because **tinyfs** does not perform any authentication or authorisation. Running as `nobody` should protect against this, as the `nobody` user should have no privileges what-so-ever (meaning it should only end up using the Others permission bits).

The final two parts is the location of the program to be run, and the arguments (the first argument here, `tinyfs`) gives the 0th argument, called `argv[0]` in the C programming language (which is the native system-level programming language on Unix-like systems), which gives the name of the program.

3. Add the following to `/etc/services` on both server and client. It must be on at least the server (as we refer to it in `/etc/inet.conf`); the client has it only for our convenience. This will let each side refer to port 900 as `tinyfs`. Note that ports are given both UDP and TCP allocations, even though they probably don't use both.

```
tinyfs   900/tcp   # Tiny File Service
tinyfs   900/udp   # Tiny File Service
```

4. Reload **inetd**.

```
# /etc/init.d/openbsd-inetd restart
```

5. Test that **lsof -Pni** shows the listening socket. Remember to use root privileges.

```
# lsof -Pni
COMMAND     PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
dhclient  …
inetd     10422 root    4u  IPv4  25808      0t0  TCP *:900 (LISTEN)
inetd     10422 root    5u  IPv6  25811      0t0  TCP *:900 (LISTEN)
```

6. Now to test it. Aim to test all cases (e.g. local and global IPv4 and IPv6 addresses) in the server.

> **Exercise**
> As an exercise try connecting using the **nc** "netcat" program, which is for IPv4 and IPv6. Recall that we installed the OpenBSD version, which supports IPv6 quite nicely. Make sure you can successfully retrieve `/etc/hostname`. Also you may try and fail to access a file such as `/etc/shadow`.

```
Loopbacks
$ echo '/etc/hostname\r\n' | nc -q-1 127.0.0.1 tinyfs
…
$ echo '/etc/hostname\r\n' | nc -q-1 ip6-localhost tinyfs
…
$ echo '/etc/hostname\r\n' | nc -q-1 ::1 900
…

Our Unique-Local Addresses…
$ echo '/etc/hostname\r\n' | nc -q-1 ip6-server1 900
…

Don't forget about your link-locals…
$ echo '/etc/hostname\r\n' | nc -q-1 fe80::a00:27ff:fee3:4d42%outside tinyfs
…
```

We've not shown all addresses that our server has. There will be at least two others (remember, Server1 has at least two interfaces). It is instructive to point out that we generally don't need to support connections coming in via the Link-Local addresses. It is important not to forget about all the various addresses that we have, in relation to restricting access to a service.

6. Access Control using TCP Wrappers

As it currently stands, `tinyfs` is currently open to the world (which is to say, it has no access control with regard to which machines can connect to the service). We shall now use TCP-Wrappers to protect `tinyfs`, and then look at how TCP-Wrappers can be used to protect other services that use `libwrap`, such as **sshd**.

tcpd is a program that is used as an access-control wrapper to protect services started from **inetd**. It grants access based on on two files: `/etc/hosts.allow` and `/etc/hosts.deny`.

If you were to look at the manual page for **tcpd**, you might find the following description.

> There are two possible modes of operation: execution of **tcpd** before a service started by **inetd**, or linking a daemon with the `libwrap` shared library as documented in the hosts_access(3) manual page. Operation when started by **inetd** is as follows: whenever a request for service arrives, the **inetd** daemon is tricked into running the **tcpd** program instead of the desired server. **tcpd** logs the request and does some additional checks. When all is well, **tcpd** runs the appropriate server program and goes away.
>
> —tcpd(8)

TCP-Wrappers will test `/etc/hosts.allow` first, and if a match is found, it will allow the connection. Otherwise, it will test `/etc/hosts.deny`, and if a match is found, it will drop the connection. Otherwise, it will allow the connection, so if you want to use TCP-Wrappers, you generally always want to ensure that a suitable deny-by-default rule is in place in `hosts.deny`.

Procedure 2. Using TCPd to Protect TinyFS

1. Start by putting the deny-by-default entry in `/etc/hosts.deny`:

```
…
ALL:ALL
```

2. Put in place a suitable rule, or set of rules, to allow the traffic you want. These rules go into `/etc/hosts.allow`:

```
…
tinyfs: 127.0.0.1 [::1] 192.168.1.0/24 [fd6b:4104:35ce::]/64
```

   The policy we have put in place here is that the server can access itself (via loopback), and our regular client ranges can access tinyfs also. Note that we haven't included the Link-Local Addresses, as these should not generally be used for applications.

   Note it is useful to pause here, and write down exactly what has effectively been *denied* entry, based on your testing cases in the previous section.

3. At this stage, `hosts.allow` and `hosts.deny` will not be consulted, because **inetd** has not yet been instructed to use **tcpd**. It is **tcpd** that checks these files, and if it allows the connection, it will pass execution (via the `exec` system call) to **tinyfs**. To do this, we change the lines regarding **tinyfs** in `inetd.conf` to the following:

```
tinyfs stream tcp4 nowait nobody /usr/sbin/tcpd /usr/local/sbin/tinyfs
tinyfs stream tcp6 nowait nobody /usr/sbin/tcpd /usr/local/sbin/tinyfs
```

4. Reload **inetd** to affect the changes you made in `inetd.conf`. Check the system logs to ensure there were no problems, and that you can see the service with an appropriate invocation of **lsof**.

5. **Exercise**

   As an exercise use **nc** (or **telnet**, which is also useful for this sort of thing) as we have done previously to connect to the service. Test all cases that should be allowed. Also test other cases that should be denied (such as the Link-Local Addresses). You should use **nc** on Client1 to try the case of `ip6-server1`. Check the system logs (in `/var/log/syslog`) reporting the actions.

Services can also be protected using `libwrap`, but we'll cover those issues when we come across the particular services, such as SSH.

Tinyfs is not for production use

Do not deceive yourself into thinking that tinyfs is ready for production use. There are many features that are still lacking, such as accounting, filtering (eg. sharing only part of the filesystem), authentication (ie. who), access control (ie. who can do what) and privacy (encrypting traffic).

7. Self-assessment

1. Ensure you have done the following successfully:

- the interfaces with appropriate interface names and IPv4 addresses;
- the right IPv4 routing table, including the default route;
- you can access the APT repository over the network;
- the IPv4 address you gave Client1 is correct;
- the **apt-get update** command working on Client1, which shows that the NAT we briefly configured is working;
- the correct IPv6 interface details for Client1 after we set up router advertisements;
- the correct IPv6 routing table for Client1 after we set up router advertisements, which should include a default route;
- the successful IPv6 ping using the Unique-Local Addresses of Server1 and Client1;
- successful testing access to **tinyfs** using **nc**, *before* protecting it with **tcpd**;
- the correct contents of the three files `hosts.allow`, `hosts.deny` and `inetd.conf`, after implementing protection using **tcpd**;
- and finally, find the log entries showing that some connections have been accepted, and others have been rejected.