Firewalls

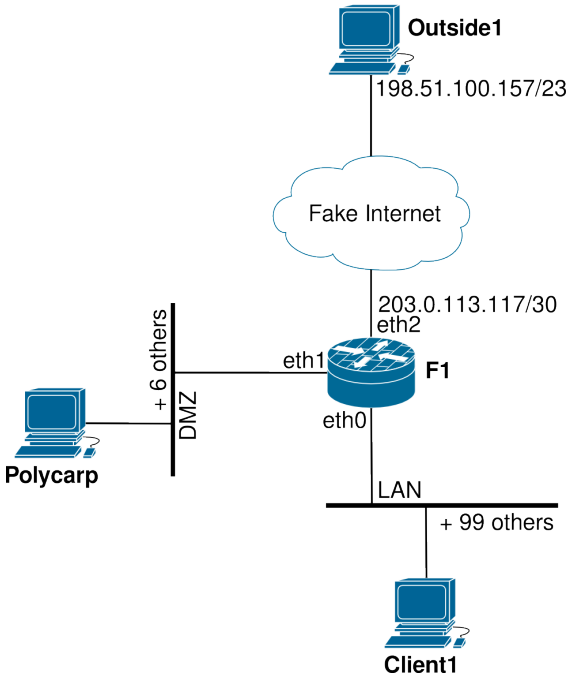
COSC301 Laboratory Manual

It is recommended that you work in pairs for this lab to make it easier to do testing and troubleshooting. Before doing this lab, you should have completed the Subnetting tutorial for IPv4, as we shall be using the same network design, as shown in Figure 1, “Network Map for Firewalling”.

You will find it very useful to have read parts of the Vyatta documentation on Firewalling, which should be available in the `resources` folder of your `Desktop` (`~/Desktop/resources/Lab Resources/Vyatta/Documentation`). Two documents are useful: `Vyatta_Firewall` and `Vyatta_NATRef`.

In this lab, you will make use of the Vyatta firewall to implement several policies suitable for a small network with a few public services. In this network, you will have an external network, an internal LAN and a De-Militarised Zone (DMZ), where your public services are being housed. As already mentioned, this is identical to the network we addressed in the Subnetting tutorial.

Figure 1. Network Map for Firewalling



The most peculiar thing about the Fake Internet here is the slight-of-hand we are playing with the default route set on Outside1 and F1. They are in different networks, but there is no routing between them, so we have played a trick whereby the packets are sent directly onto the network, as if they are delivered to a machine on the same subnet, rather than forwarding to a router. This is to make it possible for us to assign Outside1 and F1 IP addresses that are very different but connected without routers. So it looks like the Internet but without the trouble of setting up the routers. It also makes for an interesting thought experiment.

1. VirtualBox Configuration

Define the physical topology of your network; you should have learned the skills for this in the *Internal Routing* lab, but here are some reminders.

- All interfaces will be Internal Network adaptors, not NAT or anything else. Remember that F1 will have three interfaces, so call each network/switch "LAN", "DMZ" and "fake_internet" appropriately.
- Create a virtual floppy disk for F1 using **dd** as you did in the *Internal Routing* lab. Remember to use the **init-floppy** command inside Vyatta in order to save. If you don't do this, your changes will only be saved in the temporary filesystem, which is volatile in the LiveCD environment; therefore you will lose your work if you don't prepare the floppy!
- You may have issues with the default network adapter. You should choose either `PCnet-Fast III` or `Paravirtualized Network` adapter.

2. Configure and Test Basic Connectivity

Configure the basic system properties and network interfaces on F1 using the skills you learned during the lab on *Internal Routing*. F1 should be created with the Vyatta LiveCD. Do not forget the detail of settings such as the PAE/NX feature of the processor and enable of serial port.

All other machines can be created with a Ubuntu desktop LiveCD.

On the Client1 and Polycarp, just use **ifconfig** and **route** to set the interface address and add a default route to F1. On Outside1, we shall need to do something a bit different to implement our Fake Internet. On Outside1, use **ifconfig** as you would normally, but use the following **route** command:

```
# route add default dev eth0
```

This basically says "if you don't have a better route available, send it *as a local delivery* onto the eth0 link". Normally, we would have expected Outside1 to send it via a router in its own subnet.[1]

We will need a similar rule for F1 as below:

```
set system gateway-address 203.0.113.117
```

This is doing essentially the same thing. Note that the gateway is simply the same IP address of F1's interface, which seems a bit odd really. In effect, it puts it out on the local link attached to the interface with that IP address.

Once you have configured all interfaces and default routes as appropriate to each system, test that each host can ping its directly attached neighbors. You will also find that ping should work between DMZ and LAN too. For example, you should be able to ping Client1 from Polycarp.

[1] Another way of achieving the Fake Internet slight-of-hand is to have the prefix-length of Outside1 and F1s' interfaces on the Fake Internet to be a /0, which puts them in the same subnet.

A ping from Client1 to Outside1 will fail, because Outside1 doesn't know it should send the (return) packets to F1, as it just puts them on the local link (our Fake Internet). Since the destination of the packets is Client1 (10.8.2.2), the ARP for Client1 will fail and so will the delivery of the return packets.

3. Implement Source-NAT

If you haven't already, skim-read the relavent sections of the Vyatta documentation on NAT. This will save you a lot of time, and enhance your understanding. Particularly, pay attention to commands **set services nat rule ...** and the configuration examples.

Configure Source NAT for LAN and DMZ to the Fake Internet. We suggest you create two Source-NAT rules: one for traffic coming from the DMZ, and another for traffic coming from the LAN.

After the configuration, use **show service nat rule ...** to double-check if the rules are correctly set.

To test, use **echo "Response from Outside1" | sudo nc -q1 -v -l 80** in one window on Outside1, **sudo tcpdump -n -i eth0** should be running in another window on Outside1, and then launch **echo "Request from Client1" | nc -q1 -v 198.51.100.157 80** in a window on Client1. This is how you can test for a TCP connection on port 80 (or any TCP port, in general). Note you will have to start the fake server each time as it only serves a single connection and then quits. You could instead wrap the dummy-server in a loop:

```
$ while true; do
>   echo "Response from Outside1" | sudo nc -q1 -v -l 80
> done
```

4. Implementing Destination-NAT

1. Configure Destination NAT for TCP services to Polycarp and test them by connecting to F1's eth2 IP address. Have **tcpdump** running in a window on Polycarp to see if traffic is getting to it. You should open the TCP ports 25, 53, and 80 of Polycarp to Outside1.

   Ensure you have a dummy-server running on the TCP port on Polycarp that you want to be connecting to.

2. Configure Destination NAT for UDP services (e.g. port 53) to Polycarp and test them.

   We can use **nc** for UDP traffic also, just add a `-u` to the options for both the server and client. Note that, for the server to listen to a udp port, you should drop the `-q1` option of **nc**.

3. Configure Destination NAT for TCP services (e.g. port 22) to Client1 and test them.

The above suggested ports are for DNS, HTTP, SMTP and SSH, which you will need when setting up the firewall in the following sections.

5. Firewall Policies

Now that we have set up the required NAT functions and provide services to the outside network, we should set up firewall rules to restrict accesses between machines/networks and to prevent intruders from breaking out of the DMZ into the LAN.

Firewall rules are not something to be made up arbitrarily, but are instead the result of a set of policies. These are the policies that we shall be implementing in the rest of the lab:

- DNS, HTTP and SMTP services are offered by Polycarp in the DMZ, and should be accessable to everyone by default.
  Normally it would be desirable to have these services on different machines to limit the effect of a breakin, but here we are more concerned with lab resources.

- To prevent intruders from breaking out of the DMZ into the LAN, nothing is permitted to be addressed *to* F1 from the DMZ.

  Similarly, nothing in the DMZ may initiate communications with hosts in the LAN.
  In general, you don't want the LAN hosts trusting the DMZ hosts.

- SSH connections, on the standard port, from the Internet are to be forwarded through the NAT to one particular host (Client1) in the LAN. In this case, Client1 is acting as a server.

- Normal hosts in the LAN will not be able to connect to the outside network. They may only make connections to other hosts in the LAN and also into the DMZ, but only for those services offered in the DMZ.
  This sort of policy is common to force users to go through some sort of proxy. It is becoming less common in a lot of networks.

In order to begin implementing these policies, you need to at least know how to match each type of traffic. As a reminder, here is how you match each type of application's traffic. You are suggested to look through `/etc/services` and find out what the *service name* is for each port.

- DNS uses UDP port 53 and also TCP port 53.

- HTTP uses TCP port 80.

- SMTP uses TCP port 25.

- SSH uses TCP port 22.

You should begin by annotating the network diagram with each service's location and where it should be allowed from. Make sure you understand how each type of traffic will cross the router (ie. does it go *through* the router, from one interface to another, or is it addressed *to* the router, or is the traffic coming *originating from* the router.

For each flow of traffic, decide whether each direction of the firewall should have an allow-by-default or drop-by-default policy.

6. Starting to Filter

For the purposes of penetration testing, add a route on Outside1 so it sends packets to F1 in order to get to the network 10.8.2.0/24. Hint: use **sudo route add -net 10.8.2.0 gw 203.0.113.117 netmask 255.255.255.0**

Test if Outside1 can address internal hosts (it will be able to, although this is undesirable). You should be able to ping Polycarp and Client1. In this section, we shall aim to prevent this, as well as implement and test the policies defined earlier.

One thing you have to realise about Vyatta firewalls is that as soon as you attach one packet filter to an interface, *all* interface firewalls change their default behaviour from allow to

drop. This means packets will get dropped (silently), which is fine enough on a production firewall, but very frustrating to beginners when you are trying to figure out why packets are getting dropped. To help with this, we shall start by defining a simple packet filter for each firewall that implements the stateful rule (to allow packets associated to/with already allowed sessions) and logs and then drops each other packet that it sees. Then we attach the filters to the firewalls of an interface once and for all. That way, we do not need to attach the filters to the interface again but can add rules to packets filters in a nice piecemeal fashion. The logs will be sufficient for us to know which packet filter we need to adjust, and we simply adjust the packet filters until the policy works.

If you don't quite understand what we are talking about here, don't worry as you will understand after you have done the rest of the lab.

Warning

This is for those with IPTables experience only.

Internally, Vyatta does it firewalling using IPTables. However, for those with IPTables experience, there is one important difference to beware of. **set action accept** does *not* translate to a **iptables ... -j ACCEPT** but rather a **iptables ... -j RETURN**. **set action drop** however, *does* translate to a **iptables ... -j DROP**.

The annoying thing about this is that for traffic that goes through the router (ie. the IPTables FORWARD chain) you need to add rules to allow traffic in the relavent “in” and “out” packet filters. The diagrams in the Vyatta Firewall documentation show clearly how packets traverse the firewall.

Each interface has three firewalls, `in`, `out` and `local`. the `in` firewall is checking the incoming traffic to the interface. `out` is checking the outgoing traffic from the interface. `local` is for the traffic destined to the local machine (i.e. F1).

F1 has three interfaces, so we will eventually need up to nine packet filters. We shall name these nine packet filters in a logical manner, following the template FW_NETWORK_FIREWALL. For example, the packet filter attached to the `in` firewall on the Fake Internet shall be called FW_INTERNET_IN.

There is a lot of tedious typing to create the rules for each filter. For that reason, it is a common technique to use scripts to generate router configurations, which you can then easily paste into a configuration session over a Serial console or SSH (or, if you really want, Telnet). For this reason, you should enable SSH on F1 allowing access from Client1, then SSH into F1 from Client1. Here’s how to do that:

```
# set service ssh protocol-version v2
# commit
```

And now, on Client1 (hey, that's the Client1 in this Firewall lab), connect to F1:

```
client1$ ssh vyatta@10.8.2.1
…
```

Now we shall generate commands to produce our nine initial packet filters by using a shell-script. Each packet filter will start with the stateful rule, and end with a log-and-drop. For testing purposes, we have added the **log enable** directives. After testing, you can manually remove the these directives.

On Client1, put the following into a script and run it. Copy the output of the script, and paste the result into a configuration session on Vyatta.

```
for net in INTERNET DMZ LAN; do
  for fw in IN OUT LOCAL; do
    echo edit firewall name FW_${net}_${fw} rule 10
    echo   set description '"Allow already accepted traffic"'
    echo   set action accept
    echo   set state established enable
    echo   set state related enable
    echo top
    echo edit firewall name FW_${net}_${fw} rule 1024
    echo   set description '"Log and Drop traffic by default"'
    echo   set action drop
    echo   set log enable
    echo top
  done
done
```

After you paste the output of these commands into your SSH session, remember to **commit** and **save**. Don't you just love scripting for automation?

Let's see the effect of our work. **show firewall** and see what has been added. However, the packet filter's aren't actually doing anything yet, because they are not attached to the firewalls of the interfaces. A lot more commands are needed, although this time, less repetitive. Use the following script to generate the commands and then run them in a configuration session of F1.

```
echo -e 'eth2 INTERNET\neth1 DMZ\neth0 LAN' | while read iface net
do
  for fw in in out local
  do
    echo set interfaces ethernet $iface firewall $fw name FW_${net}_${fw^^}
  done
done
```

The Bash substitution ${pattern^^}

The substitution ${fw^^} simply uppercases the value of the fw variable. It is not available in older versions of Bash, such as that which come with Vyatta. It is available on Ubuntu 10.04 LTS at least, which is why you need to do this on your Ubuntu machine, and not on the firewall host.

Don't forget to **show interfaces**, **commit** and then **save**.

However, now you might run into your first stumbling block. If you were connected via SSH, you may very well find that you are no-longer able to interact with the router (this might not be the case, but it did come up during testing). At the least, any new SSH connections will be dropped. If you abort any existing hung SSH connection (using **Return ~ .**) and attempt to reconnect, the connection attempt will hang. If, on the F1 console (not via SSH) you do a **run show log tail 3** on your firewall, you will find an entry such as the following (we've highlighted the more interesting parts):

```
$ run show log tail 3
timestamp vyatta kernel: [ignorable] [FW_LAN_LOCAL-1024-D] ↩
IN=eth0 OUT= MAC=src-mac:dst-mac:ether-proto SRC=10.8.2.2 DST=10.8.2.1 ↩
LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=78110 DF ↩
```

```
PROTO=TCP SPT=52099 DPT=22 WINDOW=5840 RES=0x00 SYN URGP=0
```

Before we proceed, let's check our understanding of the log message:

FW_LAN_LOCAL-1024-D
: This is formed by the packet filter name, the rule name, and the fact that it was logged in a 'drop' rule. Thus, we know exactly where in our firewall it was dropped, and which packet filter we need to modify to let it through.

IN=eth0 OUT=
: From this we can infer that it is a 'local' chain, because it is going into the router, but not out.

  In IPTables, this would appear on the 'INPUT' primary chain, rather than the 'FORWARD' or 'OUTPUT' chain.

  We can emphatically say that the connection attempt came from the LAN, which is eth0.

SRC=10.8.2.2 DST=10.8.2.1
: The source address tells us the packet *appears* to have come from Client1, and was sent to the the IP address of F1's LAN interface. Note that this could be spoofed, although most firewalls will have options turned on to detect spoofed traffic where possible.

PROTO=TCP
: This is one of the things we need to know in order to match the traffic: that it is TCP.

DPT=22
: Knowing the destination port (in this case it is TCP/22, which is SSH) is also one of the key ways we match different types of traffic.

  For TCP, the *source* port is generally a short-lived, random high-numbered port (we call it an *ephemeral* port) and as such we generally don't concern ourselves with the source port. But occassionally we do also consult the source port for some UDP traffic.

SYN
: For TCP traffic, the SYN bit is set in the header for the first packet, thus this packet is understood to be *initiating* an SSH connection to F1, from the LAN.

So now we should be able to add a rule that allows us to get past this problem:

```
# edit firewall name FW_LAN_LOCAL
# show
rule 10 is the stateful rule…
… and rule 1024 is the default drop …
… so we want our rule to be perhaps rule 20.
# edit rule 20
# set description "Allow SSH from Client1 only"
# set action accept
# set protocol tcp
# set state new enable
# set destination port ssh
# set source address 10.8.2.2
Note that we only want one host in the LAN to access F1 via SSH. For other protocols,
we would typically specify a prefix, such as 10.8.2.0/25
# commit
```

Now check that you can access F1 via SSH from Client1. Make sure you take note of the timestamp in the logs so you can ignore log messages from previous attempts.

Another way of looking for new logs

You may find it a lot more convenient if you log into another virtual terminal on F1, and run the command **tail -f /var/log/kern.log**, which is where the kernel logs are routed to on Vyatta.

The reason that this is more convenient is that you can simple type **Return** a few times in order to visually separate old messages from new messages.

Now you can go on and implement the rest of the policy, using the same techniques.

When you are finished, you may want to adjust your logging. You don't want to log everything, as it could be used as an attack vector (either to fill up hard-disk space, or to erase logs that might be stored in a ring-buffer). But you generally want to keep some record of the sorts of things you are dropping.

Testing-by-doing is the easiest way of checking if the system works, but it is not the only method you should use to audit your firewall. In particular, you should "desk-check" the configuration to see if you are missing anything, particularly to check that you haven't left out a default case. In a real-world situation you would also test the firewall rules with tools such as port-scanners (eg. **nmap**) and packet injection tools, but the latter is rather more advanced that this lab should go and the tools are not available in the default Ubuntu Live CD (although there are Live CDs that do, you might like to look around).

7. Assessment

1. Implement and test the policies listed.

2. Use **show configuration** (or **run show configuration** if you are in configuration mode). Unlike **show** inside configuration mode, it will also show default values, which is very useful when you want to see default firewall behaviour.

   This is what you want to submit for marking.