Introduction

IPT (iptables) Traverse is a bash script that takes an iptables rule and passes it to a group of table / chain combinations. I named the groups "traversals" – I stole the term from *Linux iptables Pocket Reference*.

As an example, before a packet coming into a device reaches an application, it starts in raw prerouting, moves to mangle prerouting, then nat prerouting, mangle input, nat input, then filter input. So if the table / chain combinations are added to an "inat" traversal, all table / chain combinations can be closed, then the "inat" traversal can be opened for incoming packets that match that packet flow. For me, this is more intuitive to how I think about packets moving between devices, in terms of filtering anyways. IPT Traverse also has some options that handle replies for simple rules.

Traversals

In iptables, there are 14 table / chain combinations:
raw prerouting
mangle prerouting
nat prerouting
mangle input
nat input
filter input
mangle forward
filter forward
raw output
mangle output
nat output
filter output
mangle postrouting
nat postrouting

Say you want to create a firewall for a normal computer on which you might be using a web browser. In terms of filtering, you'd mostly only be concerned with two traversals, "i" (in) and "o"

(out). Packets going out start at raw output (the 9th combination) and move to nat postrouting (the final, or 14th combination). Reply packets coming back in start at raw prerouting (1) and move to filter input (6).

I created two additional combinations by duplicating the forward chains. This was to separate packets coming in and packets leaving the forward chain, so these traversals contain up to 16 combinations. Traversals with 14 combinations are tagged with "_s" at the end, for "short".

Here's a table of some traversals:

	all	alli	allo	io	i	o	fw	fwi	fwo	allnat	pre	in	for	out	pos
R P	x	x		x	x		x	x		x	x				
M P	x	x		x	x		x	x		x	x				
N P										x					
M I	x	x		x	x					x		x			
N I										x					
F I	x	x		x	x					x		x			
M F	x	x					x	x		x			x		
F F	x	x					x	x		x			x		
M F	x		x				x		x	x			x		
F F	x		x				x		x	x			x		
R O	x		x	x		x				x				x	
M O	x		x	x		x				x				x	
N O										x					
F O	x		x	x		x				x				x	
M P	x		x	x		x	x		x	x					x
N P										x					

Traversals with "nat" added use the NAT table, which I left out of most of the traversals because there's not much reason to use the IPT Traverse script for tasks besides filtering.

To make a traversal, copy all combinations into a text file and delete the chains, then backspace out the table name for combinations that you don't want to be part of the traversal, but leave an empty line. For example, the "all" traversal looks like this:
raw
mangle

mangle

filter
mangle
filter
mangle
filter
raw
mangle

filter
mangle

(With one blank line after the last "mangle".) Name the file "all" and put it in a directory named "./files/traversals/de/". "de" stands for "default", as opposed to "gw", which stands for "gateway". "gw" traversals have twice as many lines to handle reply packets for gateways. The script parses the text file one line at a time, and, unless it finds an empty line, it matches it with the respective chain, then outputs an iptables rule for that combination. In a "files" folder I added a text file named "chains" that looks like this:
PREROUTING
PREROUTING
PREROUTING
INPUT
INPUT
INPUT
FORWARD
FORWARD
FORWARD
FORWARD
OUTPUT
OUTPUT
OUTPUT
OUTPUT
POSTROUTING
POSTROUTING

So if you want to accept incoming packets that are established or related to established connections on a certain interface:

```
$ ./ipt_traverse.sh -t de/i -r "-i internal_interface -m conntrack --ctstate
ESTABLISHED,RELATED" -j "ACCEPT"
```

The "-t" option stands for traversal. The script picks the table from the traversal file. The "i" traversal looks like this (with another 8 blank lines):

raw

mangle

mangle

filter

The "-r" option is followed by the iptables rule, and the "-j" option is followed by the iptables target. This expands to:

```
iptables -t raw -A PREROUTING -i internal_interface -m conntrack --ctstate
ESTABLISHED,RELATED -j ACCEPT
iptables -t mangle -A PREROUTING -i internal_interface -m conntrack --ctstate
ESTABLISHED,RELATED -j ACCEPT
iptables -t mangle -A INPUT -i internal_interface -m conntrack --ctstate
ESTABLISHED,RELATED -j ACCEPT
iptables -t filter -A INPUT -i internal_interface -m conntrack --ctstate
ESTABLISHED,RELATED -j ACCEPT
```

Then you would close all incoming table / chain combinations, and, if you weren't forwarding, you could close those combinations at the same time with the "alli" traversal:

```
$ ./ipt_traverse.sh -t de/alli -j "DROP"
```

This expands to:

```
iptables -t raw -A PREROUTING -j DROP
iptables -t mangle -A PREROUTING -j DROP
iptables -t mangle -A INPUT -j DROP
iptables -t filter -A INPUT -j DROP
iptables -t mangle -A FORWARD -j DROP
iptables -t filter -A FORWARD -j DROP
```

To better explain how the script works, I'll show how to build a basic firewall for a hypothetical network topology. First, I should make some disclaimers and limitations of the script before we get too into it.

Limitations

At the moment, nothing other than the script output has been verified to work. The script has been working for me for some time, but I recently designed it to make it more user-friendly, therefore adding a fair amount of brand-new bugs as well. The firewall design for the example network topology hasn't been tested either. My topology is a little bit different, but I plan to run those tests soon. I figured these glaring omissions weren't really that big of a deal considering that the script is still in the "putting-it-out-there" phase. I haven't yet decided if this will be a recurring project for me. It will depend partly on how it's received, as I was hoping to get some feedback from some of the people who know way more about firewalls and computer security than I do. It was mostly written for the purpose creating a strict but basic set of filter rules without too much inconvenience. The script works ok for filtering but could make other tasks more messy, but I'm hoping to get it sorted in the long run.

Some notes on usage.

The scripts were written on Ubuntu 18, and haven't been tested on any other operating system.

A "net_start.sh" script is included which generates network variables. Most importantly to note, this is where the usernames of the devices are stored. This is so that permissions could be restricted to a specific user in network manager, so be sure to set your actual username in this script. There's also a dummy MAC address in case you want to restrict replies on a final device to something else connected to it, like a router. If you're using a router or some other device, set the MAC address in the "net_start.sh" script. If not, you'll have to remove the MAC source firewall rule for reply packets coming from the internet into the dedicated firewall / gateway device (which I named "b" in the examples).

Rules are expanded to iptables only – generated rules can’t be used with ipv6.

The script is very picky about parsing and hasn't been debugged, so if you're trying to get a rule set working that has no explicit example in this guide, you should assume that it won't work. Use rules exactly as described in the guide. For example, when specifying a source address within a rule, use '-s 192.168.0.1' rather than '--source 192.168.0.1' – that's what the script will look for

when it parses a rule.

The script is set to 'debug mode', which, instead of running iptables directly, echoes the rules to the terminal. You can change this by setting the "debug" variable, near the top of the script, to “0”. However I would recommend leaving it as is, and after running a command, checking each rule carefully, then copying the expanded set into a new script.

There are two traversal folders, “de” and “gw”, and using one or the other can change unspecified options that determine how the scripts expand a rule.

Traversals should be named using letters only, unless it's a short traversal, in which case '_s' is added. The script looks for '_s' to determine how a rule is expanded so it might not work right if, say, a traversal is named 'all_i_s', instead of 'alli_s'.

An iptables rule is preceded by the '-r' option, followed by a rule within double quotes.

Some options that change how an individual rule is expanded are '-p', '-d', '-s'. These options must follow the rule for them to be associated with it. The script has some overly complicated and somewhat buggy instructions for deciding default options when one or more of '-t', '-p', '-d', and '-s' are not given. If a rule set doesn't come out as expected, try adding more of these options, and, of course, feel free to look at the script to figure out why a rule set didn't come out as expected.

There are a lot of traversals that come with the script and some of them are probably not correct, so be sure to check the traversal file itself if something comes out wrong.

The script comes with a simple random executable that I wrote in c. The executable parses a file named 'seed', then automatically overwrites the file with a new seed for the next use. For some (probably obvious) reason though, the seed grows by one after the first time it's used.

Summary of Example Commands

For those who don't want to read through the whole tutorial, I'll show some of the relevant examples first.

A log command:

```
$ ./ipt_traverse.sh -t de/allnat_s -j "LOG" -l "I4_ -l files/loglist_s"
```

Output:

```
iptables -t raw -A PREROUTING -j LOG --log-prefix I4_R_PR
iptables -t mangle -A PREROUTING -j LOG --log-prefix I4_M_PR
iptables -t nat -A PREROUTING -j LOG --log-prefix I4_N_PR
iptables -t mangle -A INPUT -j LOG --log-prefix I4_M_IN
iptables -t nat -A INPUT -j LOG --log-prefix I4_N_IN
iptables -t filter -A INPUT -j LOG --log-prefix I4_F_IN
iptables -t mangle -A FORWARD -j LOG --log-prefix I4_M_FO
iptables -t filter -A FORWARD -j LOG --log-prefix I4_F_FO
iptables -t raw -A OUTPUT -j LOG --log-prefix I4_R_OU
iptables -t mangle -A OUTPUT -j LOG --log-prefix I4_M_OU
iptables -t nat -A OUTPUT -j LOG --log-prefix I4_N_OU
iptables -t filter -A OUTPUT -j LOG --log-prefix I4_F_OU
iptables -t mangle -A POSTROUTING -j LOG --log-prefix I4_M_PO
iptables -t nat -A POSTROUTING -j LOG --log-prefix I4_N_PO
```

Blocking:

```
$ for add in $(cat devices/a/blocklist); do
$     ./ipt_traverse.sh -t de/alli -r "-s $add" -j "DROP"
$     ./ipt_traverse.sh -t de/allo -r "-d $add" -j "DROP"
$ done
```

Output:

```
iptables -t raw -A PREROUTING -s 0.0.0.0 -j DROP
iptables -t mangle -A PREROUTING -s 0.0.0.0 -j DROP
iptables -t mangle -A INPUT -s 0.0.0.0 -j DROP
iptables -t filter -A INPUT -s 0.0.0.0 -j DROP
iptables -t mangle -A FORWARD -s 0.0.0.0 -j DROP
iptables -t filter -A FORWARD -s 0.0.0.0 -j DROP
iptables -t mangle -A FORWARD -d 0.0.0.0 -j DROP
iptables -t filter -A FORWARD -d 0.0.0.0 -j DROP
iptables -t raw -A OUTPUT -d 0.0.0.0 -j DROP
iptables -t mangle -A OUTPUT -d 0.0.0.0 -j DROP
iptables -t filter -A OUTPUT -d 0.0.0.0 -j DROP
iptables -t mangle -A POSTROUTING -d 0.0.0.0 -j DROP
iptables -t raw -A PREROUTING -s 255.255.255.255 -j DROP
iptables -t mangle -A PREROUTING -s 255.255.255.255 -j DROP
iptables -t mangle -A INPUT -s 255.255.255.255 -j DROP
iptables -t filter -A INPUT -s 255.255.255.255 -j DROP
iptables -t mangle -A FORWARD -s 255.255.255.255 -j DROP
iptables -t filter -A FORWARD -s 255.255.255.255 -j DROP
iptables -t mangle -A FORWARD -d 255.255.255.255 -j DROP
iptables -t filter -A FORWARD -d 255.255.255.255 -j DROP
iptables -t raw -A OUTPUT -d 255.255.255.255 -j DROP
iptables -t mangle -A OUTPUT -d 255.255.255.255 -j DROP
iptables -t filter -A OUTPUT -d 255.255.255.255 -j DROP
iptables -t mangle -A POSTROUTING -d 255.255.255.255 -j DROP
```

Using "-ll" – this creates a log rule then creates the same rule with the given target:

```
$ ./ipt_traverse.sh -t de/io -r "-i lo" -s -j "ACCEPT" -ll "AC_ -l files/loglist"
```

Output:

```
iptables -t raw -A PREROUTING -i lo -j LOG --log-prefix AC_R_PR
iptables -t raw -A PREROUTING -i lo -j ACCEPT
iptables -t mangle -A PREROUTING -i lo -j LOG --log-prefix AC_M_PR
iptables -t mangle -A PREROUTING -i lo -j ACCEPT
iptables -t mangle -A INPUT -i lo -j LOG --log-prefix AC_M_IN
iptables -t mangle -A INPUT -i lo -j ACCEPT
iptables -t filter -A INPUT -i lo -j LOG --log-prefix AC_F_IN
iptables -t filter -A INPUT -i lo -j ACCEPT
iptables -t raw -A OUTPUT -o lo -j LOG --log-prefix AC_R_OU
iptables -t raw -A OUTPUT -o lo -j ACCEPT
iptables -t mangle -A OUTPUT -o lo -j LOG --log-prefix AC_M_OU
iptables -t mangle -A OUTPUT -o lo -j ACCEPT
iptables -t filter -A OUTPUT -o lo -j LOG --log-prefix AC_F_OU
iptables -t filter -A OUTPUT -o lo -j ACCEPT
iptables -t mangle -A POSTROUTING -o lo -j LOG --log-prefix AC_M_PO
iptables -t mangle -A POSTROUTING -o lo -j ACCEPT
```

Using multiple rule options:

```
$ ./ipt_traverse.sh -t de/io -r "-p tcp -i aint -o aint" -r "-s 10.40.18.248" -p
de/o -r "-d 10.76.110.131" -p de/pre -r "-d 10.40.18.248" -p de/in -r "-m mac --
mac-source 00:80:4C:F7:85:84 -m conntrack --ctstate ESTABLISHED,RELATED" -p de/i -j
"TCP_DEVA_ -l files/chains"
```

Output:

```
iptables -t raw -A PREROUTING -i aint -p tcp -d 10.76.110.131 -m mac --mac-source
00:80:4C:F7:85:84 -m conntrack --ctstate ESTABLISHED,RELATED -j TCP_DEVA_PREROUTING
iptables -t mangle -A PREROUTING -i aint -p tcp -d 10.76.110.131 -m mac --mac-
source 00:80:4C:F7:85:84 -m conntrack --ctstate ESTABLISHED,RELATED -j
TCP_DEVA_PREROUTING
iptables -t mangle -A INPUT -i aint -p tcp -d 10.40.18.248 -m mac --mac-source
00:80:4C:F7:85:84 -m conntrack --ctstate ESTABLISHED,RELATED -j TCP_DEVA_INPUT
iptables -t filter -A INPUT -i aint -p tcp -d 10.40.18.248 -m mac --mac-source
00:80:4C:F7:85:84 -m conntrack --ctstate ESTABLISHED,RELATED -j TCP_DEVA_INPUT
iptables -t raw -A OUTPUT -o aint -p tcp -s 10.40.18.248 -j TCP_DEVA_OUTPUT
iptables -t mangle -A OUTPUT -o aint -p tcp -s 10.40.18.248 -j TCP_DEVA_OUTPUT
iptables -t filter -A OUTPUT -o aint -p tcp -s 10.40.18.248 -j TCP_DEVA_OUTPUT
iptables -t mangle -A POSTROUTING -o aint -p tcp -s 10.40.18.248 -j
TCP_DEVA_POSTROUTING
```

Using the "-s" option (swap) for automatically generating rules for reply packets:

```
$ ./ipt_traverse.sh -t gw/allall_s -A "DNS_" -r "-s 208.67.222.222" -s gw_s -j
"ACCEPT" -ll "AC_ -l files/loglist"
```

Output:

```
iptables -t raw -A DNS_PREROUTING -s 208.67.222.222 -j LOG --log-prefix AC_R_PR
iptables -t raw -A DNS_PREROUTING -s 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_PREROUTING -s 208.67.222.222 -j LOG --log-prefix AC_M_PR
iptables -t mangle -A DNS_PREROUTING -s 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_INPUT -s 208.67.222.222 -j LOG --log-prefix AC_M_IN
iptables -t mangle -A DNS_INPUT -s 208.67.222.222 -j ACCEPT
iptables -t filter -A DNS_INPUT -s 208.67.222.222 -j LOG --log-prefix AC_F_IN
iptables -t filter -A DNS_INPUT -s 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_FORWARD -s 208.67.222.222 -j LOG --log-prefix AC_M_FO
```

```
iptables -t mangle -A DNS_FORWARD -s 208.67.222.222 -j ACCEPT
iptables -t filter -A DNS_FORWARD -s 208.67.222.222 -j LOG --log-prefix AC_F_FO
iptables -t filter -A DNS_FORWARD -s 208.67.222.222 -j ACCEPT
iptables -t raw -A DNS_OUTPUT -s 208.67.222.222 -j LOG --log-prefix AC_R_OU
iptables -t raw -A DNS_OUTPUT -s 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_OUTPUT -s 208.67.222.222 -j LOG --log-prefix AC_M_OU
iptables -t mangle -A DNS_OUTPUT -s 208.67.222.222 -j ACCEPT
iptables -t filter -A DNS_OUTPUT -s 208.67.222.222 -j LOG --log-prefix AC_F_OU
iptables -t filter -A DNS_OUTPUT -s 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_POSTROUTING -s 208.67.222.222 -j LOG --log-prefix AC_M_PO
iptables -t mangle -A DNS_POSTROUTING -s 208.67.222.222 -j ACCEPT
iptables -t raw -A DNS_PREROUTING -d 208.67.222.222 -j LOG --log-prefix AC_R_PR
iptables -t raw -A DNS_PREROUTING -d 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_PREROUTING -d 208.67.222.222 -j LOG --log-prefix AC_M_PR
iptables -t mangle -A DNS_PREROUTING -d 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_INPUT -d 208.67.222.222 -j LOG --log-prefix AC_M_IN
iptables -t mangle -A DNS_INPUT -d 208.67.222.222 -j ACCEPT
iptables -t filter -A DNS_INPUT -d 208.67.222.222 -j LOG --log-prefix AC_F_IN
iptables -t filter -A DNS_INPUT -d 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_FORWARD -d 208.67.222.222 -j LOG --log-prefix AC_M_FO
iptables -t mangle -A DNS_FORWARD -d 208.67.222.222 -j ACCEPT
iptables -t filter -A DNS_FORWARD -d 208.67.222.222 -j LOG --log-prefix AC_F_FO
iptables -t filter -A DNS_FORWARD -d 208.67.222.222 -j ACCEPT
iptables -t raw -A DNS_OUTPUT -d 208.67.222.222 -j LOG --log-prefix AC_R_OU
iptables -t raw -A DNS_OUTPUT -d 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_OUTPUT -d 208.67.222.222 -j LOG --log-prefix AC_M_OU
iptables -t mangle -A DNS_OUTPUT -d 208.67.222.222 -j ACCEPT
iptables -t filter -A DNS_OUTPUT -d 208.67.222.222 -j LOG --log-prefix AC_F_OU
iptables -t filter -A DNS_OUTPUT -d 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_POSTROUTING -d 208.67.222.222 -j LOG --log-prefix AC_M_PO
iptables -t mangle -A DNS_POSTROUTING -d 208.67.222.222 -j ACCEPT
```

Using IPT Traverse with an Example Network

Say you've a got a computer that you want to connect to the internet to browse the web. Say you also have a spare computer and solar panels to hook it up to so you might as well make a dedicated firewall. Call the user computer "a" and the dedicated firewall / gateway "b". Say you also have a gaming console that you sometimes want to pass through "a" so that it's encrypted by your VPN software (VPN isn't discussed here but it isn't too hard). Call the gaming console "s" for "share".

Here's how the network would be physically set up:
Connect "a", "b", and "s" to an internal switch. On "b", in addition to the internal interface, a second external interface goes straight to the web or might connect through some other devices, like a router, first.

When using an internet browser on "a", packets go out the internal interface, then the switch, then through "b", and out to the web. Reply packets come from the web, go back through "b", and then into "a". When using a shared device, the packets leave the device and go to the internal switch, then go through "a", then through "b", and out to the web. Reply packets come back through "b", then through "a", then to the switch, then back to the device.

For the internal network, pick a local address range, like 10.0.0.0/8. For the external network device on "b", the same network range could be used, but it might be more secure to pick a different one, like 172.16.0.0/12.

Firewall Design

Now that we have some context, we can design a firewall for "a". Don't take the firewall design too seriously, it's just an example to show how the script can be used. Here's the basic strategy.

Start by dropping the kinds of packets that you would most likely never want coming into your network, like those with bad flags.

Create hardware chains. For a known topology like the one in this example, there are almost always going to be at least a few things that are known about any packet, like the IP address, MAC address, and interface name.

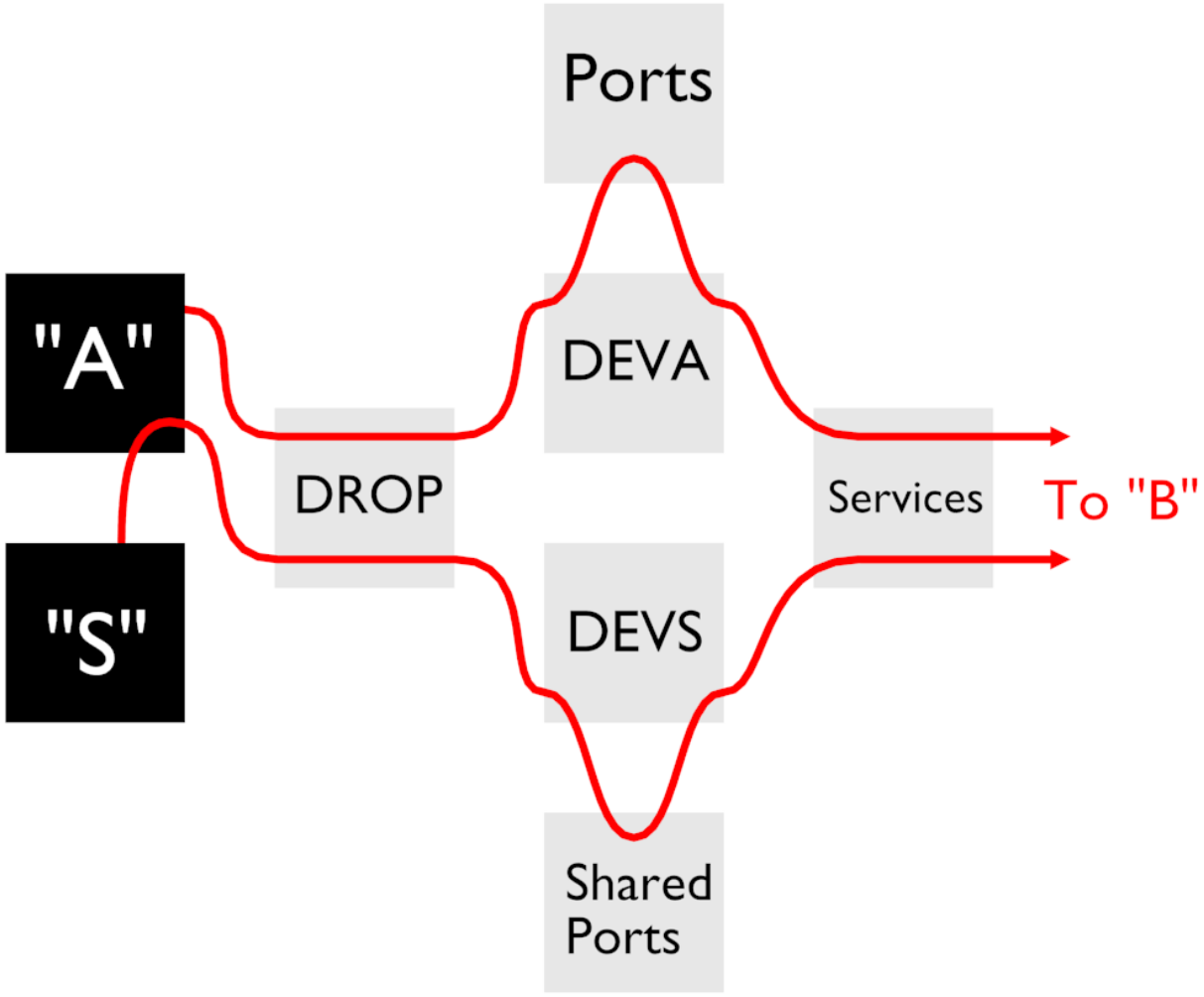
Create return port chains. Make a list of acceptable ports depending on what you want to do.

Create accept chains. Accept packets like DNS or HTTPS, depending on your needs.

Drop everything else.

Nat to help obscure internal network IPs.

The following image shows a diagram of how an accepted packet would move through “a”. The black squares represent devices and the gray squares represent chain sets.



Initializing

Before we start coding we should do some organizing. There's going to be a fair number variables that need to be incorporated into the firewall scripts, so it will be better to put them in one place rather than hard-coding.

For "a", I need an IP address, another address for SNATting, and an interface which has a name and a MAC address, and I'll be generating a random MAC for cloning. For "b", I'm using two interfaces, meaning two IPs, two MACs, and two cloned MACs. I'll be using MASQUERADE instead of SNAT, so I won't need any extra IP addresses. For "s", I have an IP address, a SNAT address, and a MAC address. There's no cloned address for this device because I'm assuming it might not be configurable.

The major downside is that I haven't yet automated a way to exchange the variables between computers, so I have to move them onto a USB drive and manually load them before running my firewall scripts.

Take a look at the "net_start.sh" script, which does several things, including generating random IPs, random MACs, and building block lists. I've configured it to enable TCP 443 and UDP 53 by default, for HTTPS and DNS. Passing arguments will enable more ports. For example, passing "HTTP" as an argument will enable TCP port 80 in addition to the default ports. To distinguish between services you want enable on "a", and services you want enabled on "s", specify the "a" services first, then pass "share", then pass the services for "s". No ports are technically open (I think) because I'm using connection tracking, but adding them to the lists prevents them from being explicitly blocked.

You also might want to look at the included random.c source which builds the random binary. It's not especially complicated and therefore might generate values that, under the right circumstances, are predictable. You can change the seed value in the "seed" file at any time, which is loaded by the random binary, and overwritten with each use.

Device “A”

With the variables created and stored in one folder, they can be loaded into a script that initiates a firewall. In the main folder, I created a new folder named "a" to hold scripts that are specific to that device. In the "a" folder I created a "start.sh" script and added the ‘#!/bin/bash’ line.

Three variables name some of the devices in the network and tell scripts which folder to look in for device variables, like IP and MAC addresses:

```
$ dev_a="a"
$ dev_b="b"
$ dev_s="s"
```

System Scripts

I start by dropping the whole network with a script named "network_state.sh". Another script named "sys.sh" sets some sysctl options. Then all policies are set to "DROP". Next a script

named "delete_and_flush.sh" clears all rules and chains from iptables and ip6tables (only to make sure ipv6 is closed).

Log Scripts

I created a folder named "log" and added the first log script. "list_all.sh" looks like this (don't forget '#!/bin/bash'):

```
$ iptables -A PREROUTING -t raw -m recent --set --name all
```

This will grab any previously unseen IP address that passes into or through the computer and puts it in a file at /proc/net/xt_recent/all. If you want to back up this file you have to do so before you shut down the computer or it's gone. If you do that it also might be a good idea to back up your current variables before the next time you run "net_start.sh", otherwise it might be harder to distinguish which IP address belongs to a device.

Now, the first IPT Traverse script, which logs all packets in each built-in table / chain combination. The script looks like this:

```
$ log_arg=$1
$ ./ipt_traverse.sh -t de/allnat_s -j "LOG" -l "$log_arg"
```

In "start.sh", the log script is loaded like this:

```
$ log_arg="I4_ -l files/loglist_s"
$ log/log_all.sh "$log_arg"
```

This expands to:

```
iptables -t raw -A PREROUTING -j LOG --log-prefix I4_R_PR
iptables -t mangle -A PREROUTING -j LOG --log-prefix I4_M_PR
iptables -t nat -A PREROUTING -j LOG --log-prefix I4_N_PR
iptables -t mangle -A INPUT -j LOG --log-prefix I4_M_IN
iptables -t nat -A INPUT -j LOG --log-prefix I4_N_IN
iptables -t filter -A INPUT -j LOG --log-prefix I4_F_IN
iptables -t mangle -A FORWARD -j LOG --log-prefix I4_M_FO
iptables -t filter -A FORWARD -j LOG --log-prefix I4_F_FO
iptables -t raw -A OUTPUT -j LOG --log-prefix I4_R_OU
iptables -t mangle -A OUTPUT -j LOG --log-prefix I4_M_OU
iptables -t nat -A OUTPUT -j LOG --log-prefix I4_N_OU
iptables -t filter -A OUTPUT -j LOG --log-prefix I4_F_OU
iptables -t mangle -A POSTROUTING -j LOG --log-prefix I4_M_PO
iptables -t nat -A POSTROUTING -j LOG --log-prefix I4_N_PO
```

"I4_" is the prefix, and the argument following the "-l" option points to a list of suffixes:
R_PR

```
M_PR
N_PR
M_IN
N_IN
F_IN
M_FO
F_FO
R_OU
M_OU
N_OU
F_OU
M_PO
N_PO
```

Make sure there's a space after each suffix, otherwise there won't be a space between the end of the log prefix and the beginning of the details of the packet.

Logging all packets might be excessive and can make a system vulnerable to flood attacks. You might want to implement the -m limit --limit and/or -m limit --limit-burst options if you have problems with your hard drive becoming full. The good news is that logs will compress themselves in the right conditions, or can be manually compressed to about 1/50th their original size in my experience.

Drop Scripts

The first drop script implements the blocklist that was created with the "net_start.sh" script. I put this script in a "drop" folder and named it “blocklist.sh”:

```
$ fil="devices/${1}/blocklist"
$ for add in $(cat "$fil"); do
$     ./ipt_traverse.sh -t de/alli -r "-s $add" -j "DROP"
$     ./ipt_traverse.sh -t de/allo -r "-d $add" -j "DROP"
$ done
```

I call it like this:

```
$ drop/blocklist.sh $dev_a
```

Say two addresses are in the blocklist, 0.0.0.0 and 255.255.255.255, each on their own line. The output is:

```
iptables -t raw -A PREROUTING -s 0.0.0.0 -j DROP
iptables -t mangle -A PREROUTING -s 0.0.0.0 -j DROP
iptables -t mangle -A INPUT -s 0.0.0.0 -j DROP
```

```
iptables -t filter -A INPUT -s 0.0.0.0 -j DROP
iptables -t mangle -A FORWARD -s 0.0.0.0 -j DROP
iptables -t filter -A FORWARD -s 0.0.0.0 -j DROP
iptables -t mangle -A FORWARD -d 0.0.0.0 -j DROP
iptables -t filter -A FORWARD -d 0.0.0.0 -j DROP
iptables -t raw -A OUTPUT -d 0.0.0.0 -j DROP
iptables -t mangle -A OUTPUT -d 0.0.0.0 -j DROP
iptables -t filter -A OUTPUT -d 0.0.0.0 -j DROP
iptables -t mangle -A POSTROUTING -d 0.0.0.0 -j DROP
iptables -t raw -A PREROUTING -s 255.255.255.255 -j DROP
iptables -t mangle -A PREROUTING -s 255.255.255.255 -j DROP
iptables -t mangle -A INPUT -s 255.255.255.255 -j DROP
iptables -t filter -A INPUT -s 255.255.255.255 -j DROP
iptables -t mangle -A FORWARD -s 255.255.255.255 -j DROP
iptables -t filter -A FORWARD -s 255.255.255.255 -j DROP
iptables -t mangle -A FORWARD -d 255.255.255.255 -j DROP
iptables -t filter -A FORWARD -d 255.255.255.255 -j DROP
iptables -t raw -A OUTPUT -d 255.255.255.255 -j DROP
iptables -t mangle -A OUTPUT -d 255.255.255.255 -j DROP
iptables -t filter -A OUTPUT -d 255.255.255.255 -j DROP
iptables -t mangle -A POSTROUTING -d 255.255.255.255 -j DROP
```

The forward chains are included in the traversals "alli" and "allo" to make sure that no listed address is either forwarded to or from a source or destination.

The next script, "mac_blocklist.sh" is almost the same, except there's no "allo" traversal, and a list of MACs is being blocked instead of IPs. "net_start.sh" adds the real MAC addresses of "a" and "b" to all blocklists since cloned MACs are going to be used.

Next is an iptables rpfilter script.

Then "bad_packets.sh" drops packets of certain address types and packets with bad flags, but doesn't use IPT Traverse in a way that hasn't already been shown, so there's not much reason to go over it.

The next script closes the forward chains if the "share" variable is not set to “1”. The "share" variable is set by "net_start.sh", when “share” is passed as an argument, which is then followed by the arguments that correspond to “services” you want enabled. This adds new ports to the shared list, and sets the "share" variable “1”. Shared ports are kept separate from ports that are available to the "a" device. In fact, the way I've set up the start script closes the input and output chains to "a's" interface whenever sharing is enabled, but this can easily be changed if you wanted to use both devices at the same time. The "forward_close.sh" script looks like this...:

```
$ ./ipt_traverse.sh -t de/for_s -j "DROP"
```

...and produces this output:

```
iptables -t mangle -A FORWARD -j DROP
iptables -t filter -A FORWARD -j DROP
```

The script is implemented like this:

```
$ if [[ "$(cat devices/a/share)" == "0" ]]; then
$     drop/forward_close.sh
$     drop/drop_external_local_in.sh $dev_a
$ elif [[ "$(cat devices/a/share)" != "0" ]]; then
$     drop/forward_only.sh
$ fi
```

Another script is named "drop_external_local_out.sh", and prevents a connection to any destination with an address that's in a local range.

The next script is a single iptables rule that drops incoming packets marked as “INVALID" by the conntrack module.

Loopback

"lo.sh" uses an option not previously mentioned: -ll. This instructs IPT Traverse to make two rules, first a log rule, then the rule given as is:

```
$ ./ipt_traverse.sh -t de/io -r "-i lo" -s -j "ACCEPT" -ll "AC_ -l files/loglist"
```

This outputs:

```
iptables -t raw -A PREROUTING -i lo -j LOG --log-prefix AC_R_PR
iptables -t raw -A PREROUTING -i lo -j ACCEPT
iptables -t mangle -A PREROUTING -i lo -j LOG --log-prefix AC_M_PR
iptables -t mangle -A PREROUTING -i lo -j ACCEPT
iptables -t mangle -A INPUT -i lo -j LOG --log-prefix AC_M_IN
iptables -t mangle -A INPUT -i lo -j ACCEPT
iptables -t filter -A INPUT -i lo -j LOG --log-prefix AC_F_IN
iptables -t filter -A INPUT -i lo -j ACCEPT
iptables -t raw -A OUTPUT -o lo -j LOG --log-prefix AC_R_OU
iptables -t raw -A OUTPUT -o lo -j ACCEPT
iptables -t mangle -A OUTPUT -o lo -j LOG --log-prefix AC_M_OU
iptables -t mangle -A OUTPUT -o lo -j ACCEPT
iptables -t filter -A OUTPUT -o lo -j LOG --log-prefix AC_F_OU
iptables -t filter -A OUTPUT -o lo -j ACCEPT
iptables -t mangle -A POSTROUTING -o lo -j LOG --log-prefix AC_M_PO
iptables -t mangle -A POSTROUTING -o lo -j ACCEPT
```

Hardware Scripts

The next scripts take packets that originate from a known device on the network and move them into a new set of chains. This is a way to "familiarize" a firewall with the information that we know will always be true about a network, such as interface names, IP addresses, and MAC addresses. You can also specify conditions that you always want to be true about certain packets, like specifying that incoming packets should always be coming through an established connection. One set of chains, separated by protocol, is created for packets that originate on "a", and another set is created for those that originate on "s". Make a script named "new_chains.sh" and put it in the main folder:

```
$ chain_arg="$1"
$ ./ipt_traverse.sh -t de/all_s -N "$chain_arg"
```

Call the script in "start.sh":

```
$ ./new_chains.sh "TCP_DEVA_"
$ ./new_chains.sh "UDP_DEVA_"
```

The first call to the “new_chains.sh” script produces the output:

```
iptables -t raw -N TCP_DEVA_PREROUTING
iptables -t mangle -N TCP_DEVA_PREROUTING
iptables -t mangle -N TCP_DEVA_INPUT
iptables -t filter -N TCP_DEVA_INPUT
iptables -t mangle -N TCP_DEVA_FORWARD
iptables -t filter -N TCP_DEVA_FORWARD
iptables -t raw -N TCP_DEVA_OUTPUT
iptables -t mangle -N TCP_DEVA_OUTPUT
iptables -t filter -N TCP_DEVA_OUTPUT
iptables -t mangle -N TCP_DEVA_POSTROUTING
```

Now packets that are known to originate from "a" can be moved into the new chains. Let's make a list of assertions that we want to be true about all packets leaving or entering the "a" device that also originated on "a" (as opposed to those that came from "s"):

1. All packets leaving and entering the computer should be doing so on the same interface.
2. All packets leaving the computer have the same source address.
3. All reply packets entering the device in the PREROUTING chains have a SNATed destination address that was already generated by "net_start.sh". (Addresses are de-SNATed in nat prerouting, the third combination.)
4. All reply packets entering the computer after the PREROUTING chain have the same destination address as the source address of packets leaving the computer.
5. All reply packets entering the computer should have the cloned MAC address of the internal interface on "b" (also generated by "net_start.sh").

6. All reply packets entering the computer should be a part of an established connection, or related to a connection that's established.

I'll first create a rule for each assertion:

```
1. -r "-i $int -o $ext"
2. -r "-s $ip1"
3. -r "-d $ip2"
4. -r "-d $ip1"
5. -r "-m mac --mac-source $mar"
6. -r "-m conntrack --ctstate ESTABLISHED,RELATED"
```

IPT Traverse has a "-p" (pattern) option that allows a traversal to be passed to a single rule argument (again: always add this after the -r and quoted rule), so I'll add a traversal for each rule:

```
1. -r "-i $int -o $ext" -p de/io
2. -r "-s $ip1" -p de/o
3. -r "-d $ip2" -p de/pre
4. -r "-d $ip1" -p de/in
5. -r "-m mac --mac-source $mar" -p de/i
6. -r "-m conntrack --ctstate ESTABLISHED,RELATED" -p de/i
```

The "-p" on the first argument can be removed because it matches the overall traversal, and the last two rules can be combined. Then add "-p tcp" to the first rule:

```
$ ./ipt_traverse.sh -t de/io -r "-p tcp -i $int -o $ext" -r "-s $ip1" -p de/o -r "-
d $ip2" -p de/pre -r "-d $ip1" -p de/in -r "-m mac --mac-source $mar -m conntrack
--ctstate ESTABLISHED,RELATED" -p de/i -j "TCP_DEVA_ -l files/chains"
```

This expands to:

```
iptables -t raw -A PREROUTING -i aint -p tcp -d 10.76.110.131 -m mac --mac-source
00:80:4C:F7:85:84 -m conntrack --ctstate ESTABLISHED,RELATED -j TCP_DEVA_PREROUTING
iptables -t mangle -A PREROUTING -i aint -p tcp -d 10.76.110.131 -m mac --mac-
source 00:80:4C:F7:85:84 -m conntrack --ctstate ESTABLISHED,RELATED -j
TCP_DEVA_PREROUTING
iptables -t mangle -A INPUT -i aint -p tcp -d 10.40.18.248 -m mac --mac-source
00:80:4C:F7:85:84 -m conntrack --ctstate ESTABLISHED,RELATED -j TCP_DEVA_INPUT
iptables -t filter -A INPUT -i aint -p tcp -d 10.40.18.248 -m mac --mac-source
00:80:4C:F7:85:84 -m conntrack --ctstate ESTABLISHED,RELATED -j TCP_DEVA_INPUT
iptables -t raw -A OUTPUT -o aint -p tcp -s 10.40.18.248 -j TCP_DEVA_OUTPUT
iptables -t mangle -A OUTPUT -o aint -p tcp -s 10.40.18.248 -j TCP_DEVA_OUTPUT
iptables -t filter -A OUTPUT -o aint -p tcp -s 10.40.18.248 -j TCP_DEVA_OUTPUT
iptables -t mangle -A POSTROUTING -o aint -p tcp -s 10.40.18.248 -j
TCP_DEVA_POSTROUTING
```

Now copy the command and change "tcp" to "udp" in the first rule and in the target.

Load the variables at the beginning of the script:

```
$ int="$(cat devices/${1}/int)"
```

```
$ ext="$(cat devices/${1}/ext)"
$ ip1="$(cat devices/${1}/ip1)"
$ ip2="$(cat devices/${1}/ip2)"
$ mar="$(cat devices/${2}/ints)"
```

This script will be unique to this device, so put it in the "a" folder.

The previous script is a little bit different than writing rules in iptables, since one command follows a packet from its origin to its termination.

Now chains can be added to the set of "DEVA" chains that return packets on ports that match the list made with "net_start.sh". Create the chains in start.sh:

```
$ ./new_chains.sh "TCP_PORTS_"
$ ./new_chains.sh "UDP_PORTS_"
```

Move packets from the "DEVA" chains to the new chains:

```
$ ./tcp_to_chains.sh "TCP_DEVA_" "TCP_PORTS_"
$ ./udp_to_chains.sh "UDP_DEVA_" "UDP_PORTS_"
```

Load the port list into the port chains:

```
$ ./ports.sh de/alli de/allo
```

Close the chains:

```
$ ./close.sh "TCP_PORTS_"
$ ./close.sh "UDP_PORTS_"
```

Now a new set of chains can be created for the "s" device. Create a list of assertions:

1. Interface name.
2. Packets coming into raw prerouting of "a" have the MAC address of the shared device.
3. Packets moving towards the internet have the source IP address of the shared device.
4. Reply packets entering the computer in PREROUTING have a SNATed destination address that was generated by "net_start.sh".
5. Reply packets after the PREROUTING chains have the same destination address as the source address of packets moving out towards the internet.
6. Reply packets coming into "a" have the cloned MAC address of the internal interface on "b".
7. Reply packets should be part of an established connection, or related to a connection that's established.

Then make one rule at a time:

```
1. -r "-i $int -o $ext"
2. -r "-m mac --mac-source $mac"
3. -r "-s $ip1"
4. -r "-d $ip2"
5. -r "-d $ip1"
6. -r "-m mac --mac-source $mar"
7. -r "-m state --state ESTABLISHED,RELATED"
```

The traversals are different since "a" is now acting as a gateway. Source packets move "through" "a", skipping the INPUT chains, moving through the FORWARD chains, then skip the OUTPUT chains. Then reply packets do the same:

```
1. -r "-i $int -o $ext" -p gw/fwfw
2. -r "-m mac --mac-source $mac" -p de/alli
3. -r "-s $ip1" -p de/fw
4. -r "-d $ip2" -p gw/prerep
5. -r "-d $ip1" -p gw/fwposnatrep
6. -r "-m mac --mac-source $mar" -p gw/allirep
7. -r "-m state --state ESTABLISHED,RELATED" -p gw/fwrep
```

Notice that the MAC source rules are receiving traversals that apply to input chains, but the main traversal will take precedence, so no input rules will be generated.

Putting it together:

```
$ ./ipt_traverse.sh -t gw/fwfw -r "-i $int -o $ext" -r "-m mac --mac-source $mac" -
p de/alli -r "-s $ip1" -p de/fw -r "-d $ip2" -p gw/prerep -r "-d $ip1" -p
gw/fwposnatrep -r "-m mac --mac-source $mar" -p gw/allirep -r "-m conntrack --
ctstate ESTABLISHED,RELATED" -p gw/fwrep -j "TCP_FW_ -l files/chains"
```

After the protocol is added to the first rule:

```
iptables -t raw -A PREROUTING -i aint -p tcp -m mac --mac-source -s 10.92.145.225 -
j TCP_FW_PREROUTING
iptables -t mangle -A PREROUTING -i aint -p tcp -m mac --mac-source -s
10.92.145.225 -j TCP_FW_PREROUTING
iptables -t mangle -A FORWARD -i aint -p tcp -m mac --mac-source -s 10.92.145.225 -
j TCP_FW_FORWARD
iptables -t filter -A FORWARD -i aint -p tcp -m mac --mac-source -s 10.92.145.225 -
j TCP_FW_FORWARD
iptables -t mangle -A FORWARD -o aint -p tcp -s 10.92.145.225 -j TCP_FW_FORWARD
iptables -t filter -A FORWARD -o aint -p tcp -s 10.92.145.225 -j TCP_FW_FORWARD
iptables -t mangle -A POSTROUTING -o aint -p tcp -s 10.92.145.225 -j
TCP_FW_POSTROUTING
iptables -t raw -A PREROUTING -i aint -p tcp -d 10.217.97.69 -m mac --mac-source
00:80:4C:F7:85:84 -m conntrack --ctstate ESTABLISHED,RELATED -j TCP_FW_PREROUTING
iptables -t mangle -A PREROUTING -i aint -p tcp -d 10.217.97.69 -m mac --mac-source
00:80:4C:F7:85:84 -m conntrack --ctstate ESTABLISHED,RELATED -j TCP_FW_PREROUTING
iptables -t mangle -A FORWARD -i aint -p tcp -d 10.92.145.225 -m mac --mac-source
```

```
00:80:4C:F7:85:84 -m conntrack --ctstate ESTABLISHED,RELATED -j TCP_FW_FORWARD
iptables -t filter -A FORWARD -i aint -p tcp -d 10.92.145.225 -m mac --mac-source
00:80:4C:F7:85:84 -m conntrack --ctstate ESTABLISHED,RELATED -j TCP_FW_FORWARD
iptables -t mangle -A FORWARD -o aint -p tcp -d 10.92.145.225 -m conntrack --
ctstate ESTABLISHED,RELATED -j TCP_FW_FORWARD
iptables -t filter -A FORWARD -o aint -p tcp -d 10.92.145.225 -m conntrack --
ctstate ESTABLISHED,RELATED -j TCP_FW_FORWARD
iptables -t mangle -A POSTROUTING -o aint -p tcp -d 10.92.145.225 -m conntrack --
ctstate ESTABLISHED,RELATED -j TCP_FW_POSTROUTING
```

Then do one more command replacing “tcp” with “udp” and add the variables to the top of "devs.sh":

```
$ int="$(cat devices/${1}/int)"
$ ext="$(cat devices/${1}/ext)"
$ ip1="$(cat devices/${2}/ip1)"
$ ip2="$(cat devices/${2}/ip2)"
$ mac="$(cat devices/${2}/mac)"
$ mar="$(cat devices/${3}/ints)"
```

The whole share block in "start.sh":

```
$ if [[ "$(cat devices/a/share)" != "0" ]]; then
$     ./new_chains.sh "TCP_DEVS_"
$     ./new_chains.sh "UDP_DEVS_"
$     a/devs.sh $dev_a $dev_s $dev_b
$     ./new_chains.sh "TCP_SHARED_PORTS_"
$     ./new_chains.sh "UDP_SHARED_PORTS_"
$     ./tcp_to_chains.sh "TCP_DEVS_" "TCP_SHARED_PORTS_"
$     ./udp_to_chains.sh "UDP_DEVS_" "UDP_SHARED_PORTS_"
$     ./ports_share.sh de/alli de/allo
$     ./close.sh "TCP_SHARED_PORTS_"
$     ./close.sh "UDP_SHARED_PORTS_"
$     drop/drop_external_local_in.sh $dev_a
$ fi
```

I made a new port script and renamed it to "ports_share.sh" so that it can load a port list that’s specific to device "s".

Accept Scripts

The first accept scripts show an example of how to implement OpenDNS, which I would barely recommend if you're encrypting your DNS requests. Make new chains:

```
$ ./new_chains.sh "DNS_"
```

Add a "services" folder with a script named "dns.sh":

```
$ loglist="files/loglist"
$ ./ipt_traverse.sh -t "$1" -A "DNS_" -r "-s 208.67.222.222" -s "$2" -j "ACCEPT" -
ll "AC_ -l $loglist"
```

```
$ ./ipt_traverse.sh -t "$1" -A "DNS_" -r "-s 208.67.220.220" -s "$2" -j "ACCEPT" -
ll "AC_ -l $loglist"
$ ./ipt_traverse.sh -t de/all_s -A "DNS_" -j "DROP"
```

I'll show the output of the first line when passing "gw/allall_s" and "gw_s" as arguments:

```
iptables -t raw -A DNS_PREROUTING -s 208.67.222.222 -j LOG --log-prefix AC_R_PR
iptables -t raw -A DNS_PREROUTING -s 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_PREROUTING -s 208.67.222.222 -j LOG --log-prefix AC_M_PR
iptables -t mangle -A DNS_PREROUTING -s 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_INPUT -s 208.67.222.222 -j LOG --log-prefix AC_M_IN
iptables -t mangle -A DNS_INPUT -s 208.67.222.222 -j ACCEPT
iptables -t filter -A DNS_INPUT -s 208.67.222.222 -j LOG --log-prefix AC_F_IN
iptables -t filter -A DNS_INPUT -s 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_FORWARD -s 208.67.222.222 -j LOG --log-prefix AC_M_FO
iptables -t mangle -A DNS_FORWARD -s 208.67.222.222 -j ACCEPT
iptables -t filter -A DNS_FORWARD -s 208.67.222.222 -j LOG --log-prefix AC_F_FO
iptables -t filter -A DNS_FORWARD -s 208.67.222.222 -j ACCEPT
iptables -t raw -A DNS_OUTPUT -s 208.67.222.222 -j LOG --log-prefix AC_R_OU
iptables -t raw -A DNS_OUTPUT -s 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_OUTPUT -s 208.67.222.222 -j LOG --log-prefix AC_M_OU
iptables -t mangle -A DNS_OUTPUT -s 208.67.222.222 -j ACCEPT
iptables -t filter -A DNS_OUTPUT -s 208.67.222.222 -j LOG --log-prefix AC_F_OU
iptables -t filter -A DNS_OUTPUT -s 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_POSTROUTING -s 208.67.222.222 -j LOG --log-prefix AC_M_PO
iptables -t mangle -A DNS_POSTROUTING -s 208.67.222.222 -j ACCEPT
iptables -t raw -A DNS_PREROUTING -d 208.67.222.222 -j LOG --log-prefix AC_R_PR
iptables -t raw -A DNS_PREROUTING -d 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_PREROUTING -d 208.67.222.222 -j LOG --log-prefix AC_M_PR
iptables -t mangle -A DNS_PREROUTING -d 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_INPUT -d 208.67.222.222 -j LOG --log-prefix AC_M_IN
iptables -t mangle -A DNS_INPUT -d 208.67.222.222 -j ACCEPT
iptables -t filter -A DNS_INPUT -d 208.67.222.222 -j LOG --log-prefix AC_F_IN
iptables -t filter -A DNS_INPUT -d 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_FORWARD -d 208.67.222.222 -j LOG --log-prefix AC_M_FO
iptables -t mangle -A DNS_FORWARD -d 208.67.222.222 -j ACCEPT
iptables -t filter -A DNS_FORWARD -d 208.67.222.222 -j LOG --log-prefix AC_F_FO
iptables -t filter -A DNS_FORWARD -d 208.67.222.222 -j ACCEPT
iptables -t raw -A DNS_OUTPUT -d 208.67.222.222 -j LOG --log-prefix AC_R_OU
iptables -t raw -A DNS_OUTPUT -d 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_OUTPUT -d 208.67.222.222 -j LOG --log-prefix AC_M_OU
iptables -t mangle -A DNS_OUTPUT -d 208.67.222.222 -j ACCEPT
iptables -t filter -A DNS_OUTPUT -d 208.67.222.222 -j LOG --log-prefix AC_F_OU
iptables -t filter -A DNS_OUTPUT -d 208.67.222.222 -j ACCEPT
iptables -t mangle -A DNS_POSTROUTING -d 208.67.222.222 -j LOG --log-prefix AC_M_PO
iptables -t mangle -A DNS_POSTROUTING -d 208.67.222.222 -j ACCEPT
```

The new option is "-s" for "swap", which automatically reverses some parts of a rule that are considered to have a direction. Currently, these include:

Interface: -i and -o.

Address: -s and -d.

Port direction: --sport and –dport.

Unfortunately, the script will only parse these options correctly if they're given in the exact form shown above.

The swap point in this case is determined by the swap file "gw_s", which is located in ./files/swaps/ and has 28 lines, with the word "swap" on line 15. (The "word" can be anything as long as the line is not blank.) Be sure to use short swap files with short traversals and long swap files with long traversals.

If you looked at above output and thought that the rule set works well for "s" but creates holes in "a" by allowing source addresses in output and destination addresses in input, take a look at the next script named "ports_to_chains.sh", which moves packets from "a" into the DNS chains:

```
$ fil="$6"
$ declare -i i
$ i=1
$ while (( $i < 5 )); do
$     line="$(source utility/get_line.sh $i $fil)"
$     if [[ "$line" != "" ]]; then
$           if (( $i == 1 )); then
$                 for sport in $line; do
$                       ./ipt_traverse.sh -t "$1" -A "$3" -r "-p tcp --sport
$sport" -j "$5 -l files/chains"
$                 done
$           fi
$           if (( $i == 2 )); then
$                 for dport in $line; do
$                       ./ipt_traverse.sh -t "$2" -A "$3" -r "-p tcp --dport
$dport" -j "$5 -l files/chains"
$                 done
$           fi
$           if (( $i == 3 )); then
$                 for sport in $line; do
$                       ./ipt_traverse.sh -t "$1" -A "$4" -r "-p udp --sport
$sport" -j "$5 -l files/chains"
$                 done
$           fi
$           if (( $i == 4 )); then
$                 for dport in $line; do
$                       ./ipt_traverse.sh -t "$2" -A "$4" -r "-p udp --dport
$dport" -j "$5 -l files/chains"
$                 done
$           fi
$     fi
$     i=$i+1
$ done
```

In "start.sh":

```
$ services/ports_to_chains.sh de/i de/o "TCP_DEVA_" "UDP_DEVA_" "DNS_"
devices/services/dns
```

Output:

```
iptables -t raw -A UDP_DEVA_PREROUTING -p udp --sport 53 -j DNS_PREROUTING
iptables -t mangle -A UDP_DEVA_PREROUTING -p udp --sport 53 -j DNS_PREROUTING
iptables -t mangle -A UDP_DEVA_INPUT -p udp --sport 53 -j DNS_INPUT
iptables -t filter -A UDP_DEVA_INPUT -p udp --sport 53 -j DNS_INPUT
iptables -t raw -A UDP_DEVA_OUTPUT -p udp --dport 53 -j DNS_OUTPUT
iptables -t mangle -A UDP_DEVA_OUTPUT -p udp --dport 53 -j DNS_OUTPUT
iptables -t filter -A UDP_DEVA_OUTPUT -p udp --dport 53 -j DNS_OUTPUT
iptables -t mangle -A UDP_DEVA_POSTROUTING -p udp --dport 53 -j DNS_POSTROUTING
```

So packets from "a" are only moved into the DNS chains if they have the right port direction. If that doesn't work for your needs, you'll have to come up with something else, like making a new set of chains or using the "share" variable to drop input and output on "a".

The "ports_to_chains.sh" script is called again for "s":

```
$ services/ports_to_chains.sh de/fw_s de/fw_s "TCP_DEVS_" "UDP_DEVS_" "DNS_"
devices/services_shared/dns
```

Output:

```
iptables -t raw -A UDP_DEVS_PREROUTING -p udp --sport 53 -j DNS_PREROUTING
iptables -t mangle -A UDP_DEVS_PREROUTING -p udp --sport 53 -j DNS_PREROUTING
iptables -t mangle -A UDP_DEVS_FORWARD -p udp --sport 53 -j DNS_FORWARD
iptables -t filter -A UDP_DEVS_FORWARD -p udp --sport 53 -j DNS_FORWARD
iptables -t mangle -A UDP_DEVS_FORWARD -p udp --sport 53 -j DNS_FORWARD
iptables -t filter -A UDP_DEVS_FORWARD -p udp --sport 53 -j DNS_FORWARD
iptables -t mangle -A UDP_DEVS_POSTROUTING -p udp --sport 53 -j DNS_POSTROUTING
iptables -t raw -A UDP_DEVS_PREROUTING -p udp --dport 53 -j DNS_PREROUTING
iptables -t mangle -A UDP_DEVS_PREROUTING -p udp --dport 53 -j DNS_PREROUTING
iptables -t mangle -A UDP_DEVS_FORWARD -p udp --dport 53 -j DNS_FORWARD
iptables -t filter -A UDP_DEVS_FORWARD -p udp --dport 53 -j DNS_FORWARD
iptables -t mangle -A UDP_DEVS_FORWARD -p udp --dport 53 -j DNS_FORWARD
iptables -t filter -A UDP_DEVS_FORWARD -p udp --dport 53 -j DNS_FORWARD
iptables -t mangle -A UDP_DEVS_POSTROUTING -p udp --dport 53 -j DNS_POSTROUTING
```

For HTTPS, new chains are created:

```
$ ./new_chains.sh "HTTPS_"
```

A generic “accept.sh” script adds a LOG / ACCEPT rule to each chain, then closes each chain in the chain set:

```
$ loglist="files/loglist"
$ ./ipt_traverse.sh -t de/all_s -A "$1" -j "ACCEPT" -ll "AC_ -l $loglist"
$ ./ipt_traverse.sh -t de/all_s -A "$1" -j "DROP"
```

"accept.sh" is called:

```
$ services/accept.sh "HTTPS_"
```

Then, like with the DNS chains, the "ports_to_chains.sh" script parses the ports made by "net_start.sh" and causes packets matching those ports to move to the new chains where they'll be immediately logged and accepted. This seems a bit strange since the "accept.sh" script is only accepting packets on ports that have made it to this point in the firewall because the same ports were already approved. But it provides a structure for stacking additional rules, as was the case in "dns.sh".

Then, for "s":

```
$ services/ports_to_chains.sh de/fw_s de/fw_s "TCP_DEVS_" "UDP_DEVS_" "HTTPS_"
devices/services_shared/https
```

The remaining "services" are implemented in the same way:

```
$ ./new_chains.sh "HTTP_"
$ services/accept.sh "HTTP_"
$ services/ports_to_chains.sh de/i de/o "TCP_DEVA_" "UDP_DEVA_" "HTTP_"
devices/services/http
$ services/ports_to_chains.sh de/fw_s de/fw_s "TCP_DEVS_" "UDP_DEVS_" "HTTP_"
devices/services_shared/http

$ ./new_chains.sh "FTP_"
$ services/accept.sh "FTP_"
$ services/ports_to_chains.sh de/i de/o "TCP_DEVA_" "UDP_DEVA_" "FTP_"
devices/services/ftp
$ services/ports_to_chains.sh de/fw_s de/fw_s "TCP_DEVS_" "UDP_DEVS_" "FTP_"
devices/services_shared/ftp

$ ./new_chains.sh "WII_"
$ services/accept.sh "WII_"
$ services/ports_to_chains.sh de/fw_s de/fw_s "TCP_DEVS_" "UDP_DEVS_" "WII_"
devices/services_shared/wii

$ ./new_chains.sh "TRA_"
$ services/accept.sh "TRA_"
$ services/ports_to_chains.sh de/i de/o "TCP_DEVA_" "UDP_DEVA_" "TRA_"
devices/services/tra
$ services/ports_to_chains.sh de/fw_s de/fw_s "TCP_DEVS_" "UDP_DEVS_" "TRA_"
devices/services_shared/tra
```

All of the "accept" lines shown above can stay in the "start.sh" script permanently, because if the corresponding argument is not passed to "net_start.sh", the "ports_to_chains.sh" script will find a blank list and no rules will be created.

The port lists for services contain four lines. The first line is reserved for TCP source ports.

If a service uses UDP only, this line should be blank ("net_start.sh" takes care of writing rules to the correct lines but I'm mentioning it in case something goes wrong). The second line is for TCP destination ports. The third and fourth lines are for UDP source and destination ports.

The accept scripts are done. Just for fun, drop everything else:

```
$ drop/drop_all.sh
```

NAT

Changing your IP address before it leaves your network can help to obscure it from devices on the web. Pass $dev_a $dev_a to a/nat.sh:

```
$ ext="$(cat devices/${1}/ext)"
$ ip1="$(cat devices/${2}/ip1)"
$ ip2="$(cat devices/${2}/ip2)"
$ iptables -t nat -I POSTROUTING -o $ext -s $ip1 -j SNAT --to-source $ip2 --random-
fully
```

Notice this script uses '-I' to insert rather than '-A'. (I'm not exactly sure what --random-fully does but it sounded good. Type 'man iptables-extensions' into a terminal for more.)

Call it one more time for "s":

```
$ if [[ "$(cat devices/a/share)" == "1" ]]; then
$     a/nat.sh $dev_a $dev_s
$ fi
```

System Scripts

Add a few more system scripts and bring up your network to finish up:

```
$ if [[ "$(cat devices/a/share)" == "0" ]]; then
$     system/sys.sh rp ipv6
$ else
$     system/sys.sh rp forward ipv6
$ fi
$ system/network_state.sh 1
$ a/network_manager.sh $dev_a $dev_b 1> /dev/null
```

Device “B”

There's one more machine to configure. Fortunately, once you copy your scripts from "a" into a new folder named "b" you're almost half-way there since the two firewalls have the same basic

design. So why use another device at all? In the next device, the input chains are going to stay closed. Excluding some kind of firewall bypass, it will harder to compromise, in theory, since it's refusing any external communication to its internal software. Packets move "through", or "past", it, as opposed to "into" it. And if logging is implemented, those logs will be harder to compromise, further defending your "a" and "s" devices. It will also add one more NAT stage, making the IP addresses on your internal network a bit more obscure.

Here's what you need to change in comparison to "a":
Add dev_r="r" to the top if you want to select for a router mac on reply packets.
On four scripts pass $dev_b instead of $dev_a: drop/blocklist.sh, drop/mac_blocklist.sh, drop/drop_external_local_in.sh, drop/drop_external_local_out.sh.
"forward_only.sh" doesn't have to be within a condition, it should be permanently enabled.
"drop_external_local_in.sh" can be permanently enabled.
"drop_external_new.sh" can be added.
The "deva.sh" and "devb.sh" scripts are a bit different. I'll go over them in a minute.
Pass de/fw_s de/fw_s to "ports.sh" and "ports_share.sh"
Pass de/fw_s de/fw_s to all "ports_to_chains.sh" scripts
From "nat.sh" down:

```
b/nat.sh $dev_b
system/sys.sh forward
system/network_state.sh 1
b/network_manager_external.sh $dev_b "Ethernet connection 1" 1> /dev/null
b/network_manager_internal.sh $dev_b 1> /dev/null
```

The “nat.sh” script uses MASQUERADE instead of SNAT:

```
$ ext="$(cat devices/${1}/ext)"
$ iptables -t nat -A POSTROUTING -o $ext -j MASQUERADE --random
```

Now for "deva.sh" and "devb.sh". Call deva.sh after the chains are made:

```
$ b/deva.sh $dev_b $dev_a $dev_r
```

The script looks like this:

```
$ int="$(cat devices/${1}/int)"
$ ext="$(cat devices/${1}/ext)"
$ ip1="$(cat devices/${2}/ip2)"
$ mac="$(cat devices/${2}/exts)"
$ mar="$(cat devices/${3}/mac)"
$ ./ipt_traverse.sh -t gw/fwfw -r "-p tcp -i $int -o $ext" -s gw -r "-s $ip1 -m mac
--mac-source $mac" -p de/fw -r "-m mac --mac-source $mar -m conntrack --ctstate
ESTABLISHED,RELATED" -p gw/fwrep -r "-d $ip1" -p gw/fwposnatrep -j "TCP_DEVA_ -l
files/chains"
$ ./ipt_traverse.sh -t gw/fwfw -r "-p udp -i $int -o $ext" -s gw -r "-s $ip1 -m mac
```

```
--mac-source $mac" -p de/fw -r "-m mac --mac-source $mar -m conntrack --ctstate
ESTABLISHED,RELATED" -p gw/fwrep -r "-d $ip1" -p gw/fwposnatrep -j "UDP_DEVA_ -l
files/chains"
```

If share is not “0”, "devb.sh" is called:

```
$ b/devs.sh $dev_b $dev_s $dev_a $dev_r
```

The script looks like this:

```
$ int="$(cat devices/${1}/int)"
$ ext="$(cat devices/${1}/ext)"
$ ip1="$(cat devices/${2}/ip2)"
$ mac="$(cat devices/${3}/exts)"
$ mar="$(cat devices/${4}/mac)"
$ ./ipt_traverse.sh -t gw/fwfw -r "-p tcp -i $int -o $ext" -s gw -r "-s $ip1 -m mac
--mac-source $mac" -p de/fw -r "-m mac --mac-source $mar -m conntrack --ctstate
ESTABLISHED,RELATED" -p gw/fwrep -r "-d $ip1" -p gw/fwposnatrep -j "TCP_DEVS_ -l
files/chains"
$ ./ipt_traverse.sh -t gw/fwfw -r "-p udp -i $int -o $ext" -s gw -r "-s $ip1 -m mac
--mac-source $mac" -p de/fw -r "-m mac --mac-source $mar -m conntrack --ctstate
ESTABLISHED,RELATED" -p gw/fwrep -r "-d $ip1" -p gw/fwposnatrep -j "UDP_DEVS_ -l
files/chains"
```

Since MASQUERADE is used instead of SNAT, these two scripts no longer select for a destination address in the PREROUTING chains of reply packets. If you wanted to change this to make your firewall a little more strict, you could try to parse your MASQUERADEd address, or create more addresses in "net_start.sh" and use one or two for SNATting.

One More Feature of IPT Traverse

You can include “-i”, “-o”, and “-m mac --mac-source” in any rule, and the script will drop those options where they’re not allowed. The drop pattern is determined by the text files in files/directions/, where a "0" indicates outgoing, "1" indicates incoming, and "2" indicates both directions, such as in the forward chains (when using short traversals).

Future versions

If I continue this project I'm ultimately interested in generating rules based on user-specified network topology. I also have a few other small features in mind and, most importantly, there are a lot of bugs that need to be fixed.