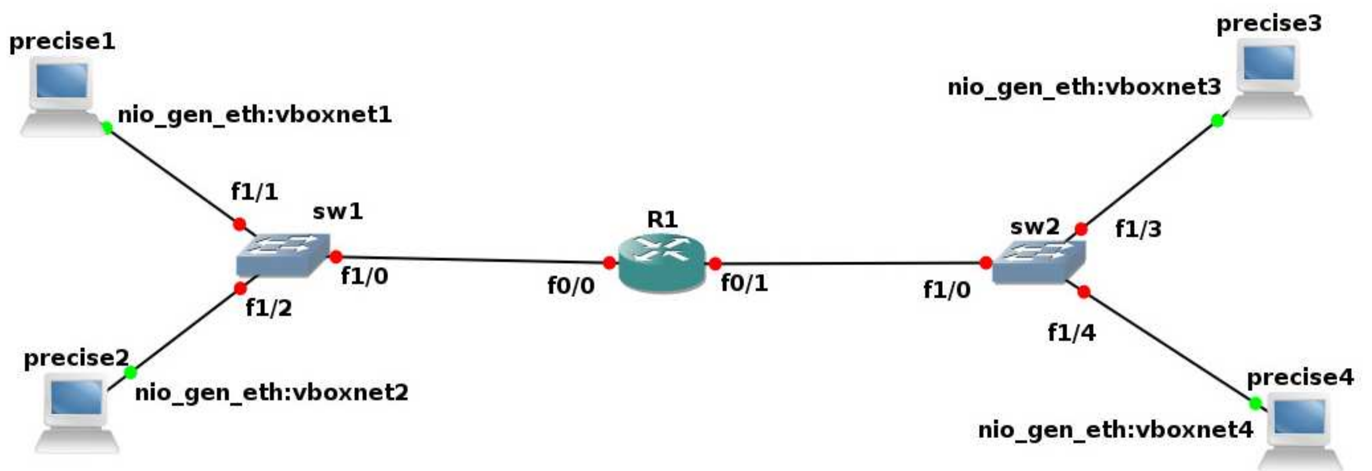
Background

ACLs



ACL

ljm

1. ACLs on routers

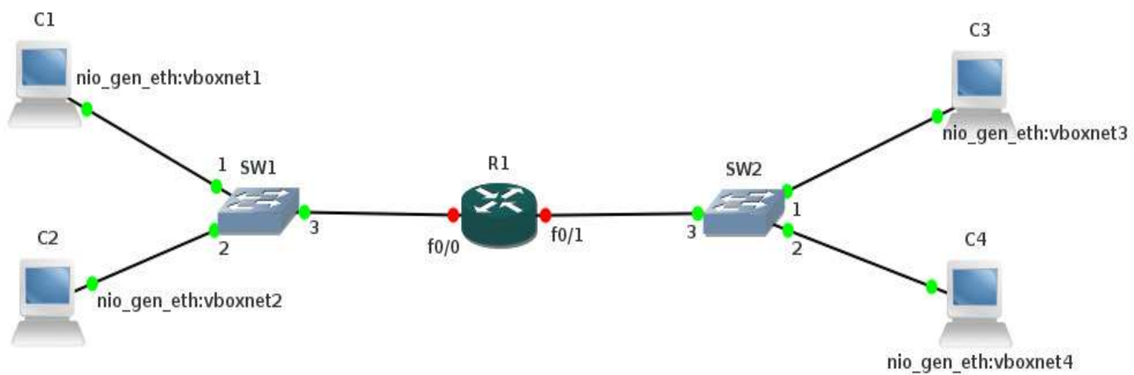
1.1 Intro

Routers are able to filter the network traffic. This is a short demonstration of how that works. We'll be using a debian machine for generating the traffic and another for answering the services.

Last run 2020-07-31
Verified: 2018-04-27.
Upgraded 2020-02-01; added line for unpredictable network adapter names in start-up scripts.

1.2 The network

We use a simple network, static routing, nothing special.



The Vagrantfile contains:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
# Vagrantfile API/syntax version. Don't touch unless you know what you're doing!
VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
        config.vm.define :xenial1 do |t|
                t.vm.box = "ubuntu/xenial64"
                t.vm.box_url = "file:////links/virt_comp/vagrant/boxes/xenial64.box"
                t.vm.provider "virtualbox" do |prov|
                        prov.customize ["modifyvm", :id, "--nic2", "hostonly", "--hos-
tonlyadapter2", "vboxnet1" ]
                end
                t.vm.provision "shell", path: "./setup.xenial1.sh"
        end
        config.vm.define :xenial2 do |t|
                t.vm.box = "ubuntu/xenial64"
                t.vm.box_url = "file:////links/virt_comp/vagrant/boxes/xenial64.box"
                t.vm.provider "virtualbox" do |prov|
                        prov.customize ["modifyvm", :id, "--nic2", "hostonly", "--hos-
tonlyadapter2", "vboxnet2" ]
                end
                t.vm.provision "shell", path: "./setup.xenial2.sh"
        end
        config.vm.define :xenial3 do |t|
                t.vm.box = "ubuntu/xenial64"
                t.vm.box_url = "file:////links/virt_comp/vagrant/boxes/xenial64.box"
                t.vm.provider "virtualbox" do |prov|
                        prov.customize ["modifyvm", :id, "--nic2", "hostonly", "--hos-
tonlyadapter2", "vboxnet3" ]
                end
                t.vm.provision "shell", path: "./setup.xenial3.sh"
        end
        config.vm.define :xenial4 do |t|
                t.vm.box = "ubuntu/xenial64"
                t.vm.box_url = "file:////links/virt_comp/vagrant/boxes/xenial64.box"
                t.vm.provider "virtualbox" do |prov|
                        prov.customize ["modifyvm", :id, "--nic2", "hostonly", "--hos-
tonlyadapter2", "vboxnet4" ]
                end
                t.vm.provision "shell", path: "./setup.xenial4.sh"
        end
end
```

and the initial router-settings are:

```
ip routing
interface FastEthernet0/0
 ip address 10.128.1.1 255.255.255.0
 no shutdown
interface FastEthernet0/1
 ip address 10.128.2.1 255.255.255.0
 shutdown
 no shutdown
```

The setup.xenial files contain the initial IP configuration.

setup.xenial1.sh:

```
ETH1=$(dmesg | grep -i 'renamed from eth1' | sed -n 's/: renamed from eth1//;s/.*
//p')
ifconfig $ETH1  10.128.1.101 netmask 255.255.255.0 up
route add -net 10.128.0.0 netmask 255.255.0.0 gw 10.128.1.1
```

setup.xenial2.sh:

```
ETH1=$(dmesg | grep -i 'renamed from eth1' | sed -n 's/: renamed from eth1//;s/.*
//p')
ifconfig $ETH1  10.128.2.100 netmask 255.255.255.0 up
route add -net 10.128.0.0 netmask 255.255.0.0 gw 10.128.2.1
```

setup.xenial3.sh:

```
ETH1=$(dmesg | grep -i 'renamed from eth1' | sed -n 's/: renamed from eth1//;s/.*
//p')
ifconfig $ETH1  10.128.1.101 netmask 255.255.255.0 up
route add -net 10.128.0.0 netmask 255.255.0.0 gw 10.128.1.1
```

setup.xenial4.sh:

```
ETH1=$(dmesg | grep -i 'renamed from eth1' | sed -n 's/: renamed from eth1//;s/.*
//p')
ifconfig $ETH1  10.128.2.101 netmask 255.255.255.0 up
route add -net 10.128.0.0 netmask 255.255.0.0 gw 10.128.2.1
```

2. Setting up the server

2.1 Goal

Our goal is to set-up a server that responds to a telnet-request on a set of ports. This will allow us to test quickly whether a service is reachable or not.

2.2 Inetd

The simplest way to create a service on Linux is to let them be started by the `inetd` On Debian, this means installing the `inetutils-inetd`. As always, it also means that we need to disable the automatic start-up at boot and launch it ourselves.

```
apt-get install inetutils-inetd
rm /etc/rc*.d/*inetutils*
```

Next, we need to set-up a simple service that allows us to see whether we can actually connect. Our simple service looks like this:

```
echo $0 $*
```

There are 10 versions of this file, called `/root/s900` to `/root/s909`.

The following lines are added to `/etc/services`

```
s900        900/tcp
s901        901/tcp
s902        902/tcp
s903        903/tcp
s904        904/tcp
s905        905/tcp
s906        906/tcp
s907        907/tcp
s908        908/tcp
s909        909/tcp
```

and we add the following lines to `/etc/inetd.conf`

```
s900    stream tcp4   nowait   root   /root/vbox/acl/s900  /root/vbox/acl/s900
s901    stream tcp4   nowait   root   /root/vbox/acl/s901  /root/vbox/acl/s901
s902    stream tcp4   nowait   root   /root/vbox/acl/s902  /root/vbox/acl/s902
s903    stream tcp4   nowait   root   /root/vbox/acl/s903  /root/vbox/acl/s903
s904    stream tcp4   nowait   root   /root/vbox/acl/s904  /root/vbox/acl/s904
s905    stream tcp4   nowait   root   /root/vbox/acl/s905  /root/vbox/acl/s905
s906    stream tcp4   nowait   root   /root/vbox/acl/s906  /root/vbox/acl/s906
s907    stream tcp4   nowait   root   /root/vbox/acl/s907  /root/vbox/acl/s907
s908    stream tcp4   nowait   root   /root/vbox/acl/s908  /root/vbox/acl/s908
s909    stream tcp4   nowait   root   /root/vbox/acl/s909  /root/vbox/acl/s909
```

The `tcp4` is the protocol name. This is a non-standard in the Debian distribution. Standard would be just `tcp` but Debian decided that that would refer to IPv6 only. Of course, we need to install the inetd:

```
apt-get -y install openbsd-inetd
```

And then, we can telnet into our new services:

```
$ vagrant ssh precise3 -c 'telnet 127.0.0.1 902'
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
/root/s902
Connection closed by foreign host.
Connection to 127.0.0.1 closed.
```

A complete setup-script for precise1 would then be:

```
ETH1=$(dmesg | grep -i 'renamed from eth1' | sed -n 's/: renamed from eth1//;s/.*
//p')
ifconfig $ETH1  10.128.1.101 netmask 255.255.255.0 up
route add -net 10.128.0.0 netmask 255.255.0.0 gw 10.128.1.1
for f in s900 s901 s902 s903 s904 s905 s906 s907 s908 s909
do
        cp /vagrant/service.sh /root/$f
        chmod a+rx /root/$f
done
cat >> /etc/services <<EOF
s900         900/tcp
s901         901/tcp
s902         902/tcp
s903         903/tcp
s904         904/tcp
s905         905/tcp
s906         906/tcp
s907         907/tcp
s908         908/tcp
s909         909/tcp
EOF
cat >>/etc/inetd.conf <<EOF
s900     stream tcp4   nowait   root   /root/s900  /root/s900
s901     stream tcp4   nowait   root   /root/s901  /root/s901
s902     stream tcp4   nowait   root   /root/s902  /root/s902
s903     stream tcp4   nowait   root   /root/s903  /root/s903
s904     stream tcp4   nowait   root   /root/s904  /root/s904
s905     stream tcp4   nowait   root   /root/s905  /root/s905
s906     stream tcp4   nowait   root   /root/s906  /root/s906
s907     stream tcp4   nowait   root   /root/s907  /root/s907
s908     stream tcp4   nowait   root   /root/s908  /root/s908
s909     stream tcp4   nowait   root   /root/s909  /root/s909
EOF
apt-get -y install openbsd-inetd
ps -ef | grep inet
```

The `ps` is there to show that the inetd works.

3. ACL Basics

3.1 Introduction

Cisco uses ACLs to filter traffic. ACLs are also used in different contexts, like for example NAT. Access lists consist of a number of permit and deny rules. ACLs are placed on an interface and there are inbound and outbound ACLs.

People tend to view a router with ACLs as a sort of firewall thing. The main difference is that a router in general does not do statefull filtering. If statefull filtering is used (Cisco calls it reflexive), it consumes a lot of resources on the router.

There are two types of ACL:

— standard: deny from source IP addresses

— extended: allow more criteria, like port numbers, destination and protocol

ACLs are numbered:

standard	1-99	1300-1999
extended	100-199	2000-2699

In general, you will make a design based on what you wantt to filter, where and why. Before you do however, you need to know the properties of the filtering and the options. It is exactly this what we want to do here.

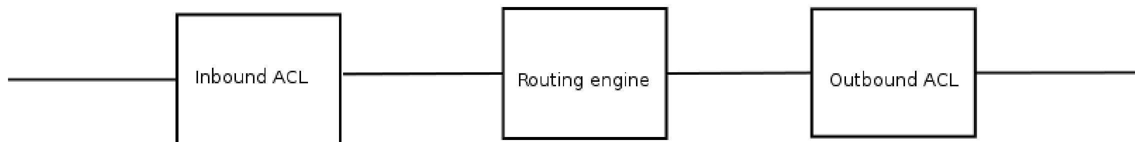
The simpler filter ACLs are, the better. Although routers can be re-configured from time to time, in general the management tooling is not made for frequent rule changes (like in firewalls).

3.2 ACL types

Filtering ACLs are connected to an interface. ACLs that are used for filtering can be

— inbound

— outbound



There are two types of ACL:

— standard: deny from source IP addresses

— extended: allow more criteria, like port numbers, destination and protocol

ACLs are named or they are numbered:

standard	1-99	1300-1999
extended	100-199	2000-2699

Standard ACLs are much more simple than extended. Standard ACLs only allow filtering of source IP. The following table gives the idea behind ACLs:

type:	numbered	named	
ID:	number	name	
configure with	global commands	sub commands	
	standard numbered	standard named	standard matching: source IP address
	extended numbered	extended named	extended matching: source & dest. IP source & dest. port other criteria

In general, the most ACLs that I have seen are extended numbered.

ACLs are followed top-to-bottom; the first matching rule defines the action.

3.3 Creating ACLs

3.3.1 Standard numbered

Matching a single IP address:

```
access-list 1 permit 10.128.1.101
```

Matching a subnet is a bit counter-intuitive. Cisco has chosen to create a "wildcard mask" for this, in stead of using the normal subnet mask. Mainly to annoy people, I presume. The wildcard mask is the inverse of the subnet mask, for all practical purposes. An example then would be:

```
access-list 2 permit 10.128.2.0 0.0.0.255
```

3.3.2 Extended numbered

Extended ACLs allow a finer control of the filtering. The syntax is:

```
access-list <number> {permit|deny} <protocol> <source> <destination> [port specifica-
tion] [other options]
```

parameter	explanation
access-list	the keyword to define the accesslist
number	the number of the ACL; 100-199 or 2000-2699 for extended ACLs
permit|deny	allow or deny ation for this rule
protocol	name of the IP protocol. Usually ip, tcp, udp or icmp.
source	can be a single host or an subnet with wildcard mask
destination	can be a single host or an subnet with wildcard mask
port specification	an operator (lt (less than), gt (greater than), eq (equal), ne (not equal) or range) with e port specification.
other options	mostly used to specify ’established’ to allow only one direction of the traffic.

The source and destination can be:

any	the any keyword matches any ip address
host <ip address>	the host-keyword, followed by an IP address matches a single host
<ip address> <wildcard mask>	matches a subnet; the wildcard mask has been described above.

3.3.3 Named ACLs
Although named ACLs should provide some more documentational advantages, I have seldom seen them being used. The definition is a bit different, but the concepts are more or less the same.

```
ip access-list standard filtername
   permit 10.128.1.101
   permit 10.128.1.102
```

Likewise, extended ACLs can be created. Functionally, the named ACLs are the same as their numbered cousins. Therefore, we'l not continue with these named ACLs.

4. Example ACLs

4.1 A standard ACL
In our test network, we will allow precise1 to access the precise3 and 4, but we will disallow precise2 that access.

source	destination	action
precise1	precise3, precise4	allow
precise2	precise3, precise4	deny

Because this is just a simple exersice, we will not place any filters for the return traffic. The ACL becomes:

```
access-list 1 permit 10.128.1.101
access-list 1 deny   10.128.1.102
```

And we will apply it to interface f0/0 incoming:

```
interface FastEthernet0/0
 ip access-group 1 in
```

To test, a simple ping will do; standard ACLs do not use protocol or port filtering.

```
[ljm@verlaine acl]$ vagrant ssh precise1 -c 'ping -c1 10.128.2.103'
PING 10.128.2.103 (10.128.2.103) 56(84) bytes of data.
64 bytes from 10.128.2.103: icmp_req=1 ttl=63 time=13.6 ms
--- 10.128.2.103 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 13.672/13.672/13.672/0.000 ms
Connection to 127.0.0.1 closed.
[ljm@verlaine acl]$ vagrant ssh precise2 -c 'ping -c1 10.128.2.103'
PING 10.128.2.103 (10.128.2.103) 56(84) bytes of data.
From 10.128.1.1 icmp_seq=1 Packet filtered
--- 10.128.2.103 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms
Connection to 127.0.0.1 closed.
[ljm@verlaine acl]$
```

Removing the ACL is done with the `no` keyword:

```
r1#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
r1(config)#int f0/0
r1(config-if)#no ip access-group 1 in
r1(config-if)#^Z
r1#conf t
r1(config)#no access-list 1
r1(config)#^Z
r1#
```

And a verification that the access-list is now gone:

```
[ljm@verlaine acl]$ vagrant ssh precise1 -c 'ping -c1 10.128.2.103'
PING 10.128.2.103 (10.128.2.103) 56(84) bytes of data.
64 bytes from 10.128.2.103: icmp_req=1 ttl=63 time=33.8 ms
--- 10.128.2.103 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 33.857/33.857/33.857/0.000 ms
Connection to 127.0.0.1 closed.
[ljm@verlaine acl]$ vagrant ssh precise2 -c 'ping -c1 10.128.2.103'
PING 10.128.2.103 (10.128.2.103) 56(84) bytes of data.
64 bytes from 10.128.2.103: icmp_req=1 ttl=63 time=28.8 ms
--- 10.128.2.103 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 28.880/28.880/28.880/0.000 ms
Connection to 127.0.0.1 closed.
[ljm@verlaine acl]$
```

4.2 Extended numbered

Extended numbered ACLs offer greater flexibility. To show this, we will allow the following trafic:

source	destination	port	action
precise1	precise3	tcp 901,tcp 902	allow
precise1	precise4	tcp 903,tcp 904	allow
precise2	precise3	tcp 905,tcp 906	allow
precise2	precise3	tcp 907,tcp 908	allow
precise3	precise1,precise2	tcp_909	allow

This table sees the router as a single filtering component between the two networks. But that is not the way that a router works. There will be differnt ACLs, based on the direction of the traffic. Also, we need to consiider on which interface the ACLs are placed. Cisco recommends:

— place extended ACLs close to the source

— place standard ACLs close to the destination But that guideline does not look at the fact that you may have a loopback address available in your router.

We'll define two ACLs, one from 10.128.1.0/24 to 10.128.2.0/24 and one the otherway around:

```
access-list 110 permit tcp host 10.128.1.101 host 10.128.2.103 eq 901
access-list 110 permit tcp host 10.128.1.101 host 10.128.2.103 eq 902
access-list 110 permit tcp host 10.128.1.101 host 10.128.2.104 eq 903
access-list 110 permit tcp host 10.128.1.101 host 10.128.2.104 eq 904
access-list 110 permit tcp host 10.128.1.102 host 10.128.2.103 eq 905
access-list 110 permit tcp host 10.128.1.102 host 10.128.2.103 eq 906
access-list 110 permit tcp host 10.128.1.102 host 10.128.2.104 eq 907
access-list 110 permit tcp host 10.128.1.102 host 10.128.2.104 eq 908
access-list 110 permit tcp any any established
access-list 111 permit tcp host 10.128.2.103 host 10.128.1.101 eq 909
access-list 111 permit tcp host 10.128.2.103 host 10.128.1.102 eq 909
access-list 111 permit tcp any any established
```

Any traffic that has the 'established'-bit set is allowed. This allows return-traffic in an existing session to pass through the router.

The access-list 110 is put on f0/0 as input list and 111 on f0/1 as input.

```
int f0/0
ip access-group 110 in
int f0/1
ip access-group 111 in
```

Verifying gives:

```
vagrant ssh precise1 -c 'telnet 10.128.2.103 901'
Trying 10.128.2.103...
Connected to 10.128.2.103.
Escape character is '^]'.
/root/s901
Connection closed by foreign host.
Connection to 127.0.0.1 closed.
[ljm@verlaine acl]$ vagrant ssh precise1 -c 'telnet 10.128.2.103 902'
Trying 10.128.2.103...
Connected to 10.128.2.103.
Escape character is '^]'.
/root/s902
Connection closed by foreign host.
Connection to 127.0.0.1 closed.
[ljm@verlaine acl]$ vagrant ssh precise1 -c 'telnet 10.128.2.103 903'
Trying 10.128.2.103...
telnet: Unable to connect to remote host: No route to host
Connection to 127.0.0.1 closed.
vagrant ssh precise3 -c 'telnet 10.128.1.101 909'
Trying 10.128.1.101...
Connected to 10.128.1.101.
Escape character is '^]'.
/root/s909
Connection closed by foreign host.
Connection to 127.0.0.1 closed.
[ljm@verlaine acl]$ vagrant ssh precise3 -c 'telnet 10.128.1.101 908'
Trying 10.128.1.101...
telnet: Unable to connect to remote host: No route to host
Connection to 127.0.0.1 closed.
```

The rest is also as expected.

5. How secure are router ACLs

5.1 Introduction

Cisco states in http://www.cisco.com/c/en/us/support/docs/ip/access-lists/13608-21.html "Devised to prevent unauthorized direct communication to network devices, infrastructure access control lists (iACLs) are one of the most critical security controls that can be implemented in networks." But how effective are those ACLs?

To see how effectiv the ACLs are, we introduce another host, Kali. This is a virtual machine with Kali linux which has an eth1 adapter on vboxnet0. This virtual machine is not provisioned by Vagrant, so we'll need to do some manual configuration:

```
ifconfig eth1 10.128.1.10 netmask 255.255.255.0
route add -net 10.128.0.0 netmask 255.255.0.0 gw 10.128.1.1
```

This is more or less the same as for precise1 and precise2. And, because it is on sw1, we can ping precise1:

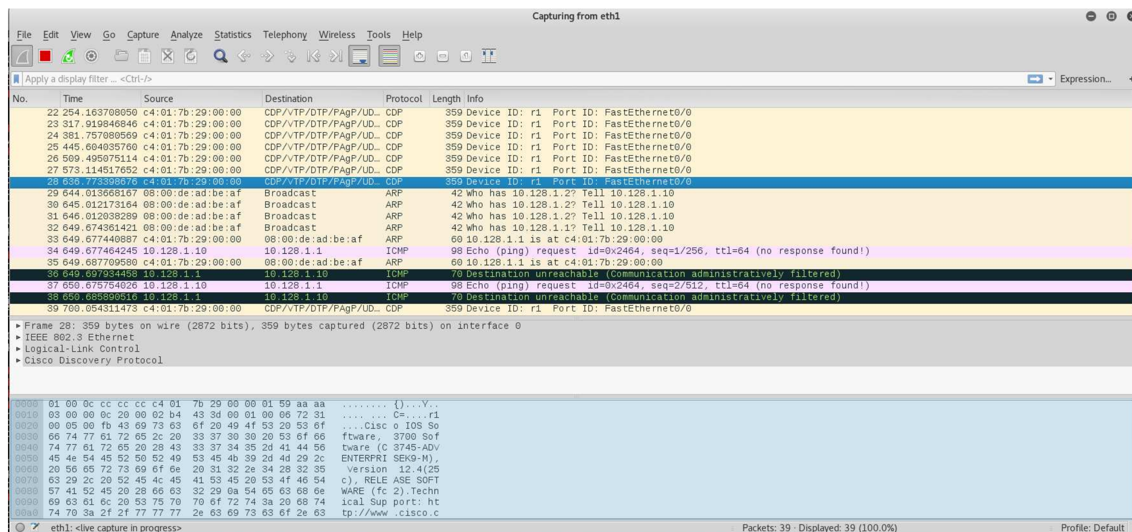
```
ping 10.128.1.101
PING 10.128.1.101 (10.128.1.101) 56(84) bytes of data.
64 bytes from 10.128.1.101: icmp_seq=1 ttl=64 time=1.38 ms
64 bytes from 10.128.1.101: icmp_seq=2 ttl=64 time=0.916 ms
64 bytes from 10.128.1.101: icmp_seq=3 ttl=64 time=0.910 ms
^C
--- 10.128.1.101 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.910/1.069/1.381/0.220 ms
```

5.2 Ping

But pinging the router from Kali must fail (we did not permit incoming ICMPs):

```
ping 10.128.1.1
PING 10.128.1.1 (10.128.1.1) 56(84) bytes of data.
From 10.128.1.1 icmp_seq=1 Packet filtered
From 10.128.1.1 icmp_seq=2 Packet filtered
From 10.128.1.1 icmp_seq=3 Packet filtered
^C
--- 10.128.1.1 ping statistics ---
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 2009ms
```

What is interesting is that we see that ping actually sees that the router is there (otherwise we would get a "destination host unreachable") The image below shows what happens. First a ping from Kali to 10.128.1.2 (a non-existing address), then a ping to 10.128.1.1 (the router).



The ping to the non-existing address does not pass the arp-phase. However, for the 10.128.1.1, the router replies to the ARP. Furthermore, the router replies with an ICMP 70 (Destination unreachable).

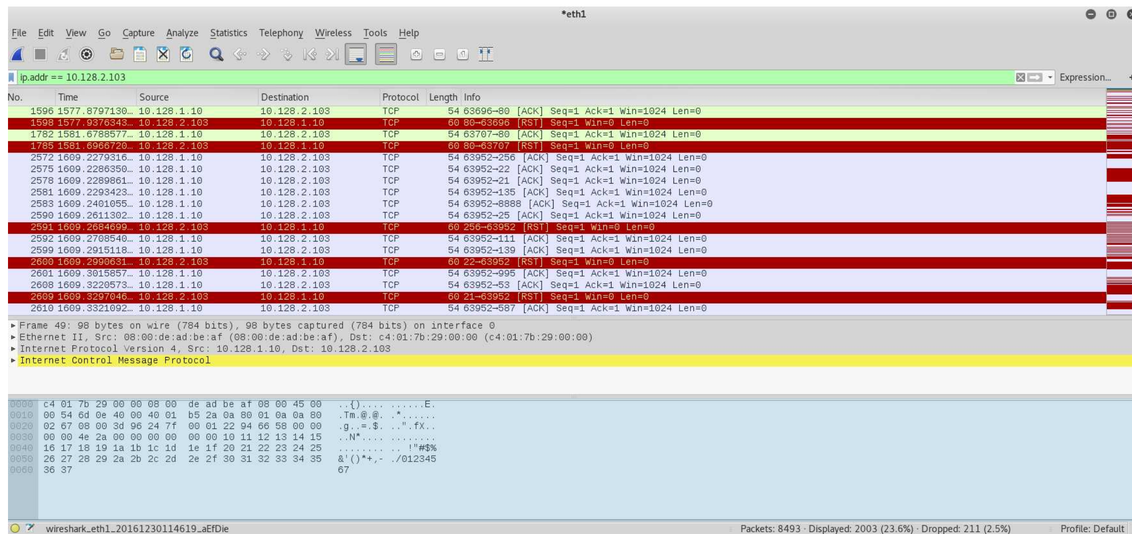
5.3 What is behind the router?

Ping is nice, but anything behind the router is invissible for the pings. The router effectively hides the network behind the router. The response for precise3 is the same as for 10.128.2.109 (a non-existing host).

Of course, no-one will be discouraged by this. Nmap is always availabe on Kali:

```
nmap -sA 10.128.2.0/24
Starting Nmap 7.25BETA2 ( https://nmap.org ) at 2016-12-30 12:12 EST
Nmap scan report for 10.128.2.1
Host is up (0.078s latency).
All 1000 scanned ports on 10.128.2.1 are unfiltered
Nmap scan report for 10.128.2.103
Host is up (0.26s latency).
All 1000 scanned ports on 10.128.2.103 are unfiltered
Nmap scan report for 10.128.2.104
Host is up (0.15s latency).
All 1000 scanned ports on 10.128.2.104 are unfiltered
Nmap done: 256 IP addresses (3 hosts up) scanned in 95.53 seconds
```

How is that possible? Wireshark shows the way this works:



We opened-up the return traffic rather wide: all established traffic is let through. You can ofcourse limit this traffic by allowing only specific return traffic. This efficitively doubles the size of your access-list. That means that you will make a trade-off between manegeability and security.

It may also be a good idea to get a firm control of any trafic that leaves the router. The recomendation to place extended ACLs close to the source is therefore perhaps not such a good idea.

However, the router is still a stateless inspection device. Each packet is examined individually; no mechanism exists to relate a packet to an existing session. This, in addition to the fact that router ACLs are not really easily managed, introduces the need for a more sophisticated device, the firewall.

CONTENTS

1. ACLs on routers 0
1.1 Intro 0
1.2 The network 0

2. Setting up the server 2
2.1 Goal 2
2.2 Inetd 2

3. ACL Basics 4
3.1 Introduction 4
3.2 ACL types 5
3.3 Creating ACLs 6

4. Example ACLs 7
4.1 A standard ACL 7
4.2 Extended numbered 8

5. How secure are router ACLs 10
5.1 Introduction 10
5.2 Ping 11
5.3 What is behind the router? 11