Blue Team Module 1:
Introduction to Networking with Netcat and Wireshark

By: Andrew Kerr <www.andrewjkerr.com>

Introduction

The Internet is one of my favorite places. You can read news, keep in touch with friends, and, most importantly, laugh at doge memes. But, how exactly does the Internet work? In this module, we'll take a look at some of the underlying technologies that make the Internet function as well as some of the tools that we can use to interact and analyze those technologies.

Module

OSI Model

Perhaps the best place to start to understand networking is the OSI model. While the OSI model is just a conceptual model, the abstraction layers allow for a basic understanding of the communication system that the internet uses. Figure 1 below presents the seven layers along with examples of each.

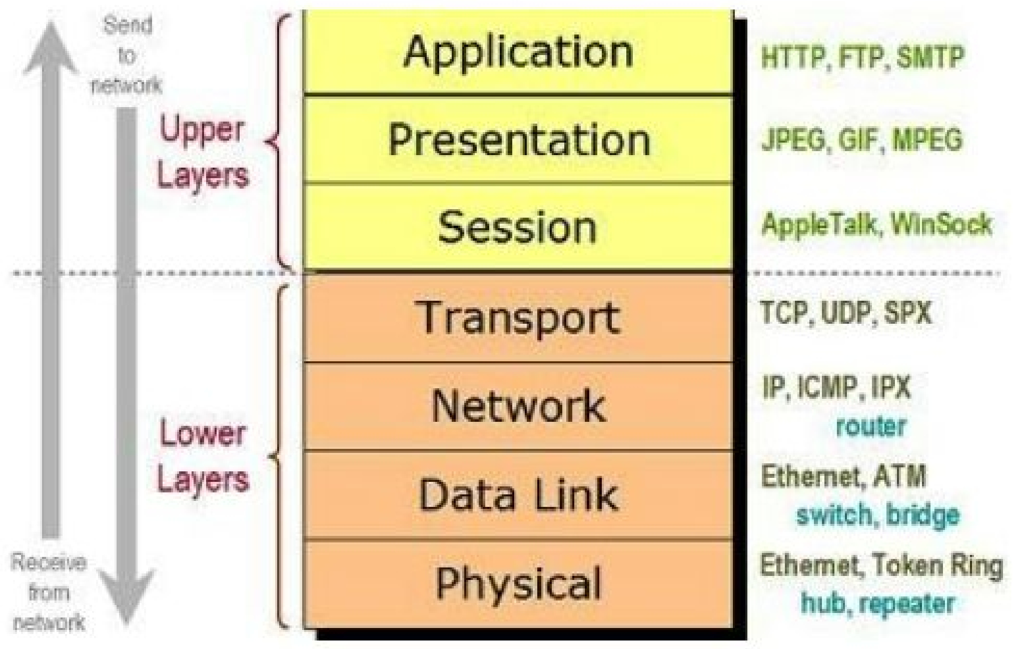


Figure 1: The OSI Model (image source: about.com)

While it is important to have a fundamental understanding of all seven layers, for the purpose of this module, we will be focusing on the Transport and Network layers.

IP Packets
Like we learned last week, computers can only understand 1s and 0s. So, when we send data from one computer to another, we need to figure out how to break up the data and make sure that all of the data actually gets there - we don't want an email that's missing an entire sentence!

In older dial-up systems, the modems used something like morse code to send data. How it would work is that the 1s and 0s were translated into sounds, like in morse code, that the phone lines could easily send and the modem would decode the tones as a 1 or a 0. Pretty neat huh?

Today, we thankfully don't use sounds or dial-up, but we still have to use 1s and 0s in order for the computer to understand the data that we're sending/receiving. And, since we're sending massive amounts of data, the issue of breaking up the data and delivering all of it to the receiving computer is even more complicated.

Enter packets. Residing in layer 3 (network) of our OSI model, a packet is just a formatted unit of data that's carried by a network that will be routed from the source to the destination. Each packet contains two types of data: control information and a payload. The control information is the information necessary to actually deliver the packet. For example, the source and destination addresses and sequencing information is included within the control information. Typically, the control information is stored in the header and/or the trailer while the payload comes in between.

While there are many different types of packets, the kind we'll be focusing on is the Internet Protocol (IP) packet - specifically IPv4. Each IP packet contains a header (Figure 2) and a payload (to be discussed later).

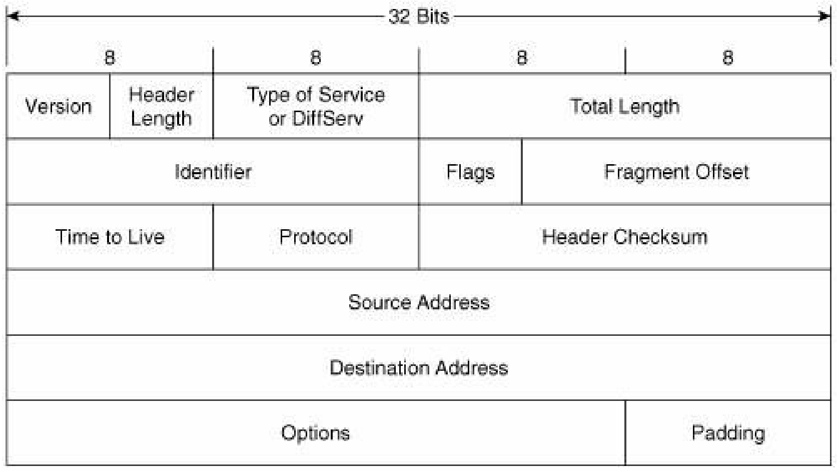


Figure 2: IP Packet Header (source: mycertstudies.com)

As you can see, each IP packet header contains the necessary control information (most importantly the source address and destination address) and will allow for the payload to get to the right place.

TCP/IP

Now that we've covered the header, we can talk about the payload. While there are a few different types of payloads we could talk about, we'll be discussing the Transmission Control Protocol (TCP) as TCP is so common that the entire IP suite is normally referred to as TCP/IP.

Remember when we said that we needed to break up the data and make sure that all of the data gets to the destination? TCP will do this for us. How TCP works is that it will accept data from a data stream, break it up into data chunks, and adds a TCP header which will create a TCP segment (Figure 3). This TCP segment is then sent as the payload of the IP packet.

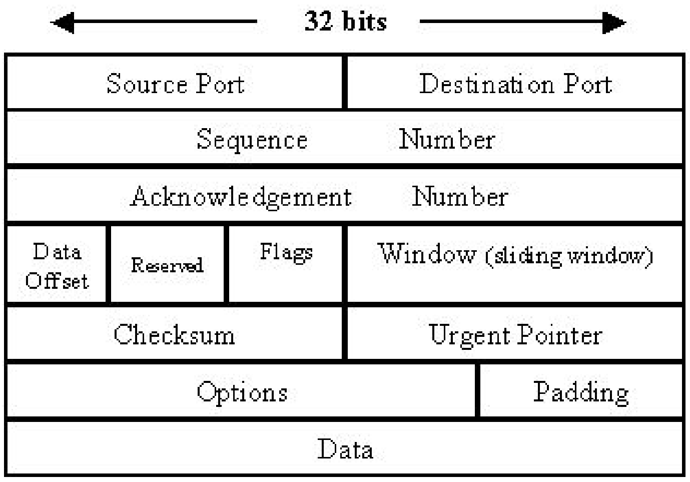


Figure 3: TCP Segment (source: techrepublic.com)

One of the most important differences between TCP and other payload protocols is that TCP is ordered and error checked. In order for our TCP segments to be delivered in order, we see a "Sequence Number" in the segment header that will allow the receiver to reconstruct the data, carried in the data section, in the correct order. When the receiver receives a TCP segment, it will send an "Acknowledgment Number" back to the sender that will be the sequence number of the next segment to send. The error checking of TCP segments is done with the "Checksum" part of the header and, if the received segment does not match the checksum being send, the receiver will ask the sender to resend that segment.

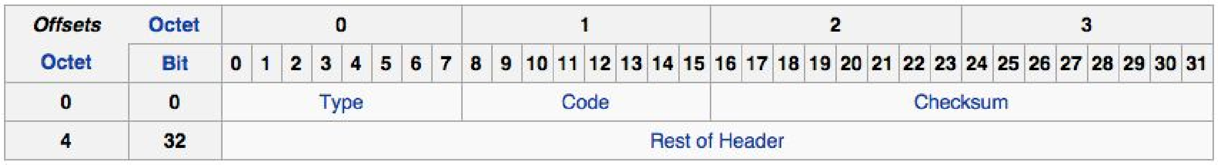
Remember how I said that TCP was really common? Well, here's how common TCP is: a *lot* of different applications/services use TCP (HTTP, SSH, FTP, SSL just to name a few). So, how does TCP know which application/service to route the segment to?

In order to introduce how this is done, think about the mailboxes for apartment complexes/dorms. Each apartment most likely has the same address, but each apartment has a different apartment number. For this example, let's assume the apartment complex is at 123 SANDERS ST and your apartment number was 456. The address 123 SANDERS ST would get your mail to your apartment complex, but you also need to specify APT #456 in order for your mail to be placed in the correct mailbox.

With TCP/IP, the IP packet only specifies the IP address (123 SANDERS ST in our example), but we need to specify which application to deliver (APT #456). We can do this with port numbers. Each TCP header contains a source and destination port number which is the Internet socket ("mailbox") that the data should be delivered to. Common port numbers are 80 (HTTP), 20/21 (FTP), 25 (SMTP), and 443 (SSL).

ICMP

Another type of payload for IP packets that is important to know is the Internet Control Message Protocol (ICMP). The biggest difference between TCP and ICMP is that ICMP is normally not used (but can be!) to exchange data between systems and is mainly used with diagnostic tools such as ping and traceroute. Since ICMP is normally not used to transmit data, the ICMP header (Figure 4) is normally the only part of the ICMP segment.



Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Type								Code								Checksum															
4	32	Rest of Header																															

Figure 4: ICMP Segment Header (source: wikipedia.org)

The important sections to pay attention to are the first two bytes: the type and code. The type identifies the control message and the code is a description of that message. For example, a type of 3 with a code of 1 means that the destination was unreachable (type) because the destination host was unreachable (code). We'll be taking a look at some ICMP segments in the next section.

Introducing Wireshark

Now that we've learned about TCP/IP and ICMP, we can start taking a look at some of the segments and packets we've been talking about using a free tool called Wireshark. Wireshark describes itself as "the world's foremost network protocol analyzer" and will allow you to analyze the data going over your network. In our case, we'll be using it to analyze ICMP (in this section) and TCP (in a later section) segments. (Just a reminder, I'll be using the VMs distributed at the first meeting.)

On Kali Linux, let's go ahead and start up Wireshark in the background by opening up a shell and using the command `*wireshark &*`. This will open up a Wireshark window as shown in Figure 5.

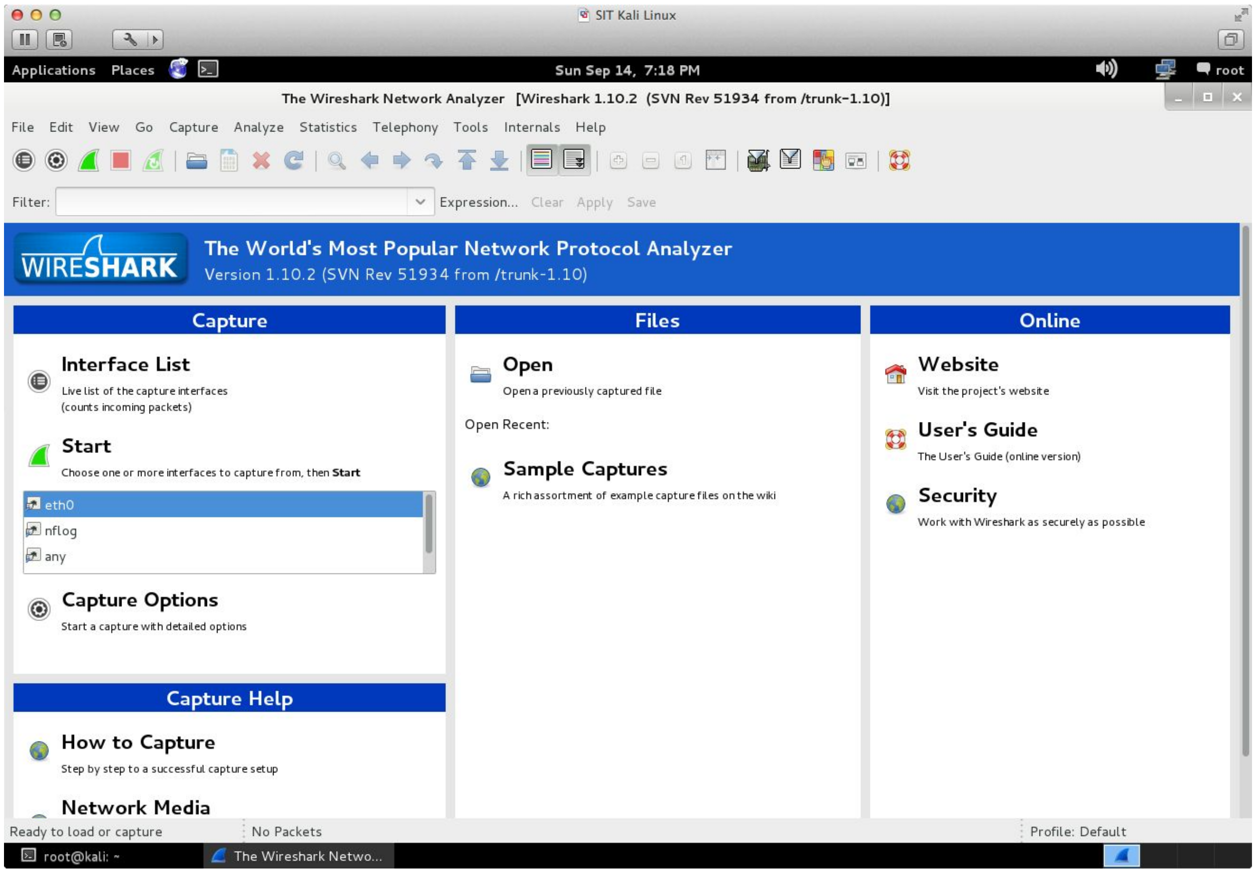

SIT Kali Linux
Applications Places
Sun Sep 14, 7:18 PM
root
The Wireshark Network Analyzer [Wireshark 1.10.2 (SVN Rev 51934 from /trunk-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: Expression... Clear Apply Save
WIRESHARK
The World's Most Popular Network Protocol Analyzer
Version 1.10.2 (SVN Rev 51934 from /trunk-1.10)
Capture
Interface List
Live list of the capture interfaces (counts incoming packets)
Start
Choose one or more interfaces to capture from, then Start
eth0
nflog
any
Capture Options
Start a capture with detailed options
Capture Help
How to Capture
Step by step to a successful capture setup
Network Media
Files
Open
Open a previously captured file
Open Recent:
Sample Captures
A rich assortment of example capture files on the wiki
Online
Website
Visit the project's website
User's Guide
The User's Guide (online version)
Security
Work with Wireshark as securely as possible
Ready to load or capture
No Packets
Profile: Default
root@kali: ~
The Wireshark Netwo...

Figure 5: Wireshark Start Screen

Now that we have started Wireshark, let's start our first capture. Select eth0 from the interface list under the "Start" button and then press "Start". You should see a window like in Figure 6.

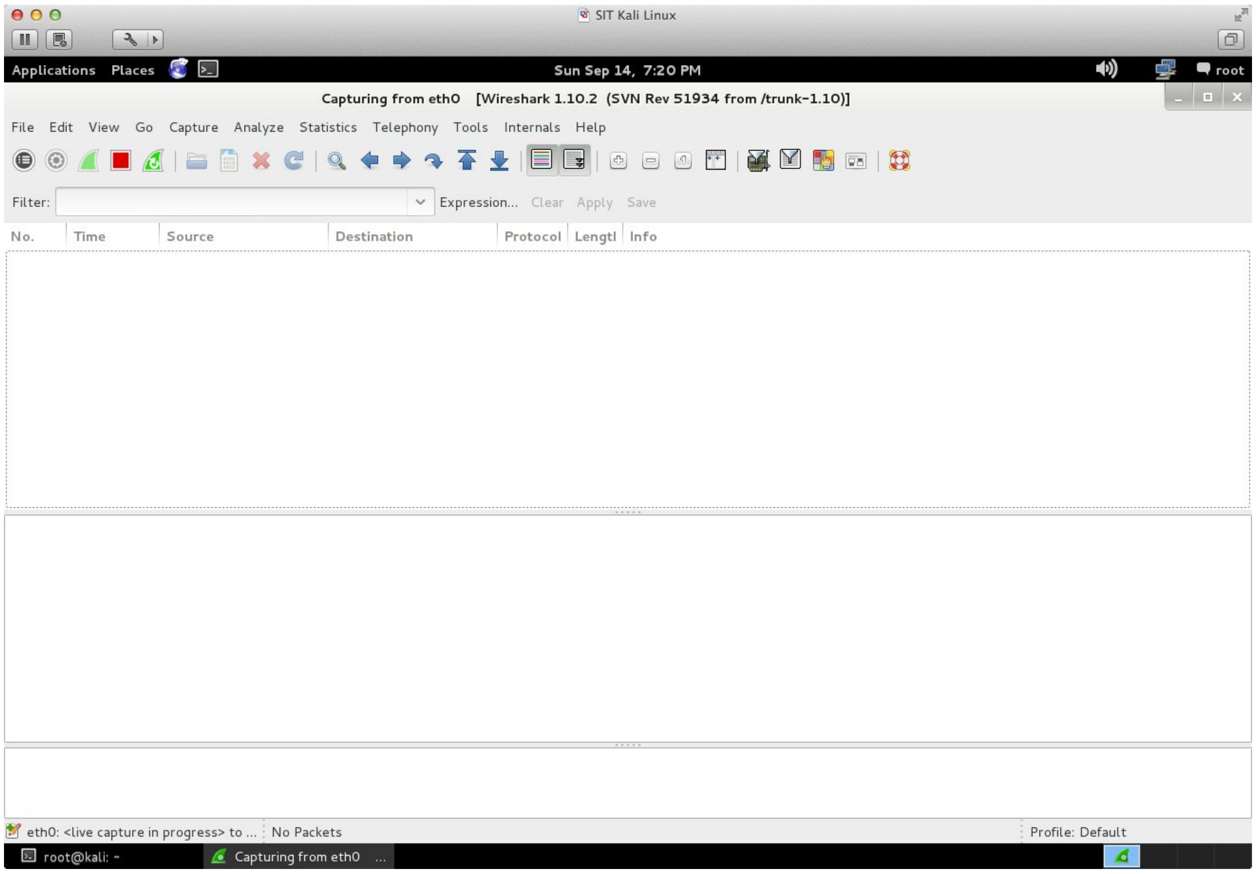
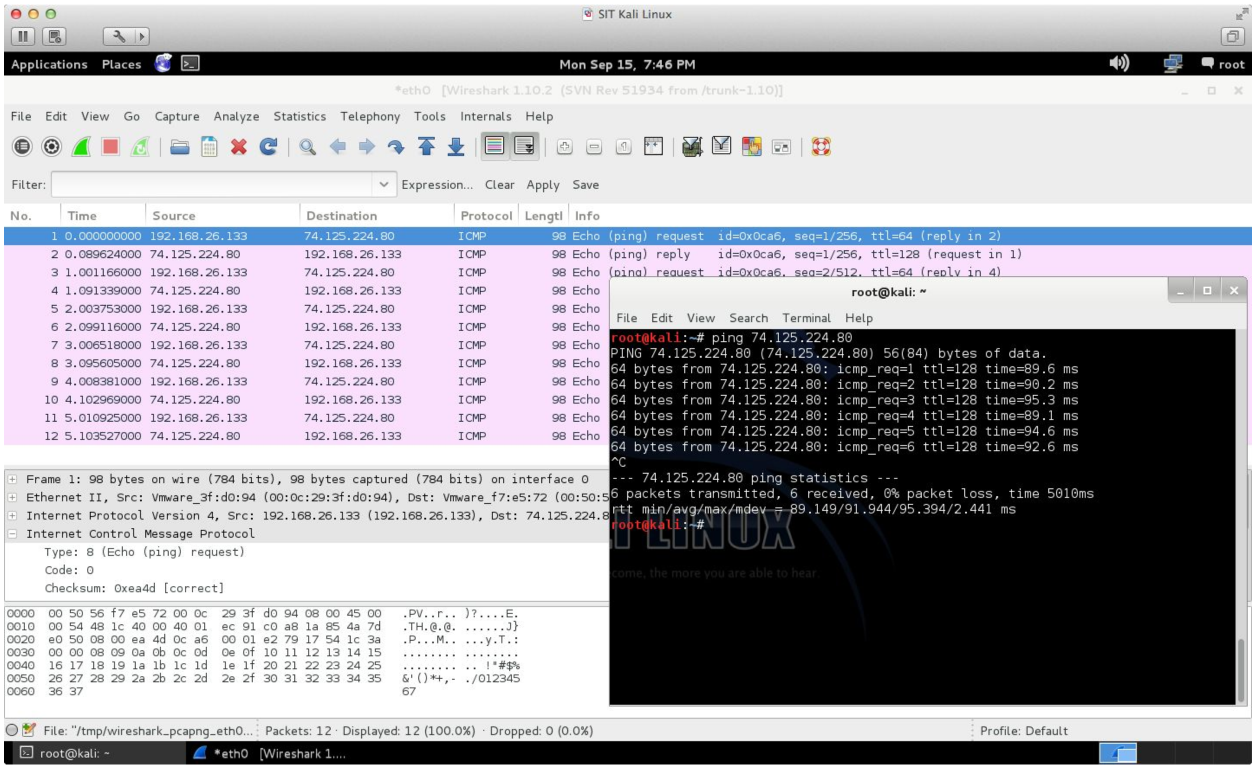

SIT Kali Linux
Applications Places
Sun Sep 14, 7:20 PM
root
Capturing from eth0 [Wireshark 1.10.2 (SVN Rev 51934 from /trunk-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save
No. Time Source Destination Protocol Lengtl Info
eth0: <live capture in progress> to ...
No Packets
Profile: Default
root@kali: ~
Capturing from eth0 ...

Figure 6: Wireshark Capture Window

Let's get started by looking at some ICMP segments. Open up your terminal again and type `*ping 74.125.224.80*` and let it run for a few seconds before killing it with ctrl+c. You should see something like in Figure 7.


SIT Kali Linux
Applications
Places
Mon Sep 15, 7:46 PM
root
*eth0 [Wireshark 1.10.2 (SVN Rev 51934 from /trunk-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save
root@kali: ~
File Edit View Search Terminal Help
Profile: Default

```
No.  Time        Source          Destination     Protocol Length Info
   1 0.000000000 192.168.26.133  74.125.224.80   ICMP     98 Echo (ping) request  id=0x0ca6, seq=1/256, ttl=64 (reply in 2)
   2 0.089624000 74.125.224.80   192.168.26.133  ICMP     98 Echo (ping) reply    id=0x0ca6, seq=1/256, ttl=128 (request in 1)
   3 1.001166000 192.168.26.133  74.125.224.80   ICMP     98 Echo (ping) request  id=0x0ca6, seq=2/512, ttl=64 (reply in 4)
   4 1.091339000 74.125.224.80   192.168.26.133  ICMP     98 Echo
   5 2.003753000 192.168.26.133  74.125.224.80   ICMP     98 Echo
   6 2.099116000 74.125.224.80   192.168.26.133  ICMP     98 Echo
   7 3.006518000 192.168.26.133  74.125.224.80   ICMP     98 Echo
   8 3.095605000 74.125.224.80   192.168.26.133  ICMP     98 Echo
   9 4.008381000 192.168.26.133  74.125.224.80   ICMP     98 Echo
  10 4.102969000 74.125.224.80   192.168.26.133  ICMP     98 Echo
  11 5.010925000 192.168.26.133  74.125.224.80   ICMP     98 Echo
  12 5.103527000 74.125.224.80   192.168.26.133  ICMP     98 Echo
```

```
+ Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
+ Ethernet II, Src: Vmware_3f:d0:94 (00:0c:29:3f:d0:94), Dst: Vmware_f7:e5:72 (00:50:5
+ Internet Protocol Version 4, Src: 192.168.26.133 (192.168.26.133), Dst: 74.125.224.8
- Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0xea4d [correct]
```

```
0000  00 50 56 f7 e5 72 00 0c  29 3f d0 94 08 00 45 00   .PV..r.. )?....E.
0010  00 54 48 1c 40 00 40 01  ec 91 c0 a8 1a 85 4a 7d   .TH.@.@. ......J}
0020  e0 50 08 00 ea 4d 0c a6  00 01 e2 79 17 54 1c 3a   .P...M.. ...y.T.:
0030  00 00 08 09 0a 0b 0c 0d  0e 0f 10 11 12 13 14 15   ........ ........
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ........ .. !"#$%
0050  26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35   &'()*+,- ./012345
0060  36 37                                              67
```

```
File: "/tmp/wireshark_pcapng_eth0... Packets: 12 · Displayed: 12 (100.0%) · Dropped: 0 (0.0%)
```

```
root@kali:~# ping 74.125.224.80
PING 74.125.224.80 (74.125.224.80) 56(84) bytes of data.
64 bytes from 74.125.224.80: icmp_req=1 ttl=128 time=89.6 ms
64 bytes from 74.125.224.80: icmp_req=2 ttl=128 time=90.2 ms
64 bytes from 74.125.224.80: icmp_req=3 ttl=128 time=95.3 ms
64 bytes from 74.125.224.80: icmp_req=4 ttl=128 time=89.1 ms
64 bytes from 74.125.224.80: icmp_req=5 ttl=128 time=94.6 ms
64 bytes from 74.125.224.80: icmp_req=6 ttl=128 time=92.6 ms
^C
--- 74.125.224.80 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5010ms
rtt min/avg/max/mdev = 89.149/91.944/95.394/2.441 ms
root@kali:~#
```

Figure 7: Our Wireshark capture from ping

Now, if we look at our packet capture we can see two IPs sending packets back and forth - 192.168.26.133 and 74.125.224.80. If we look closer, we can see in the info column that, when 192.168.26.133 was the source, it was an "Echo (ping) request" and, when 74.125.224.80 was the source, it was an "Echo (ping) reply". How does Wireshark know about whether it was a request or a reply?

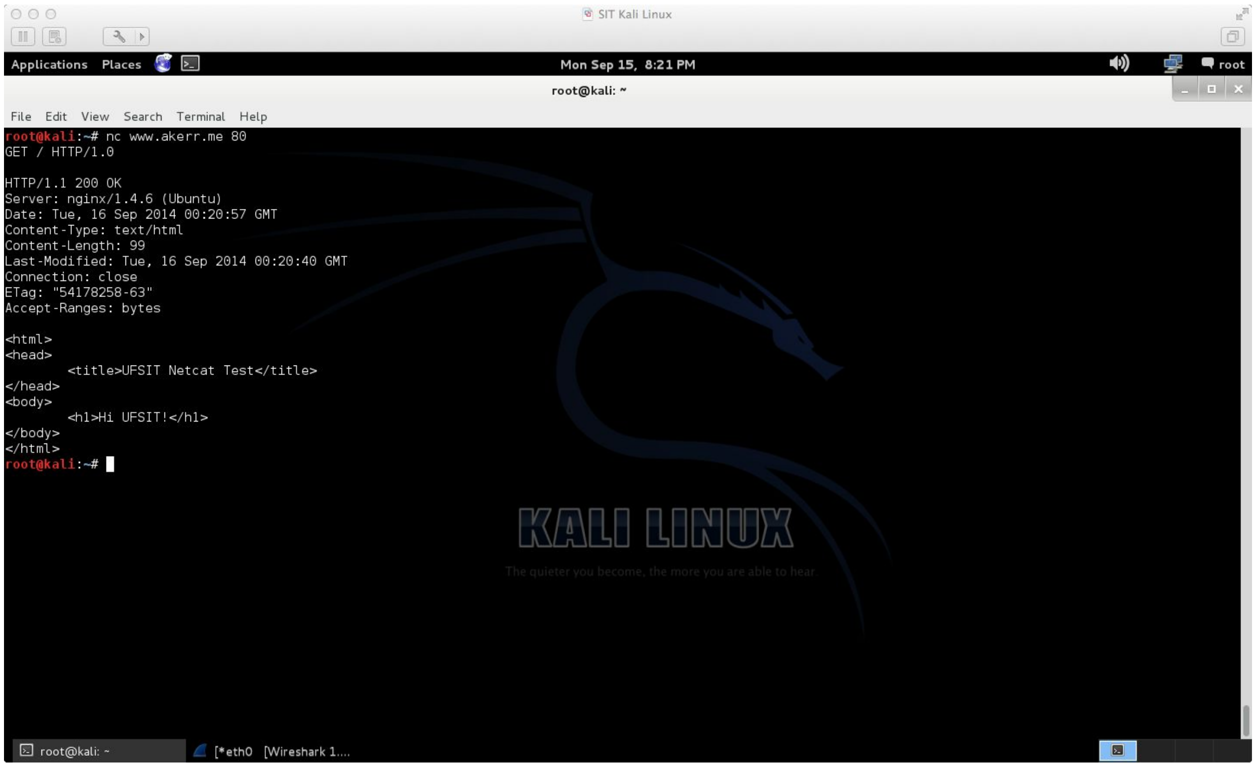
Well, remember when we talked about the types and codes? That's how! If you look in the panel in the middle that has a dropdown for "Internet Control Message Protocol" and expand the content, you can see that Wireshark has extracted the information from the ICMP header and displayed it in an easy to read fashion. Now that we've gotten acquainted with Wireshark, let's move onto another very important tool - Netcat.

Introducing Netcat

As stated in the introduction, Netcat is the "tcp/ip swiss army knife" because it has so much functionality in regards to TCP/IP. There have been a lot of different reimplementations, such as Ncat from the Nmap Project, but we will be using Netcat for this document.

To demonstrate how Netcat works, let's open up a shell and type `*nc www.akerr.me 80*` and press enter. You should have been bumped down to a newline which means that Netcat is now waiting for input to send to the connected server. So, what should we say?

Well, if we recall from above, TCP port 80 is the port for HTTP. Using Netcat, we can send a HTTP GET request to the server and see what we get back. To do this, just type `*GET / HTTP/1.0*` and press enter twice to send the request to the server. You should see something like in Figure 8.

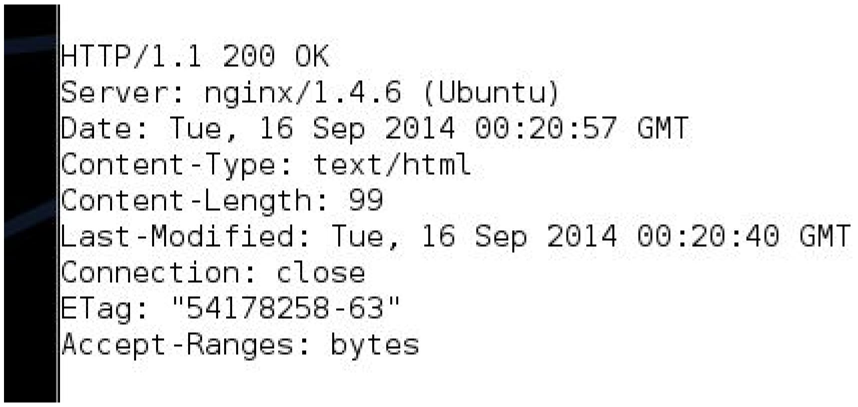


```
root@kali:~# nc www.akerr.me 80
GET / HTTP/1.0

HTTP/1.1 200 OK
Server: nginx/1.4.6 (Ubuntu)
Date: Tue, 16 Sep 2014 00:20:57 GMT
Content-Type: text/html
Content-Length: 99
Last-Modified: Tue, 16 Sep 2014 00:20:40 GMT
Connection: close
ETag: "54178258-63"
Accept-Ranges: bytes

<html>
<head>
        <title>UFSIT Netcat Test</title>
</head>
<body>
        <h1>Hi UFSIT!</h1>
</body>
</html>
root@kali:~# 
```

Figure 8: The response from the server

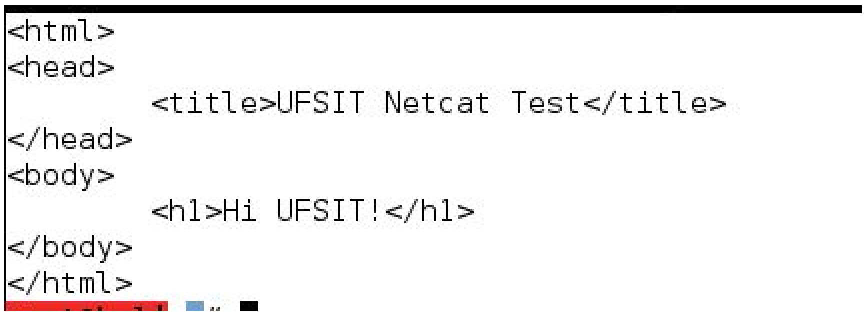
As you can see, we got a response back from the server! However, how should we interpret this?

Well, HTTP responses are broken up into a “response header” and a “response body”. The response header (Figure 9) contains the response code (in this case, it’s “200 OK”), the date, the server, and more. One important thing to note is the “content type”. For this request, the content type is “text/html” and, if we look in the response body (Figure 10), we can see it’s just HTML code!



```
HTTP/1.1 200 OK
Server: nginx/1.4.6 (Ubuntu)
Date: Tue, 16 Sep 2014 00:20:57 GMT
Content-Type: text/html
Content-Length: 99
Last-Modified: Tue, 16 Sep 2014 00:20:40 GMT
Connection: close
ETag: "54178258-63"
Accept-Ranges: bytes
```

Figure 9: The Response Headers



```
<html>
<head>
        <title>UFSIT Netcat Test</title>
</head>
<body>
        <h1>Hi UFSIT!</h1>
</body>
</html>
```

Figure 10: The Response Body

What we just did with Netcat was essentially what a web browser does when it visits a site using HTTP: connects to the server over port 80, does a GET request, and displays the response body. And, sure enough, if we go to www.akerr.me we see that the response body is there (Figure 11)!

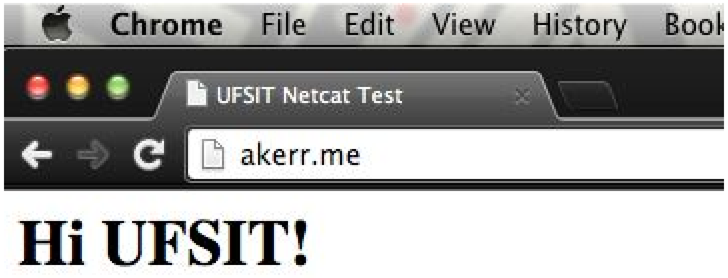


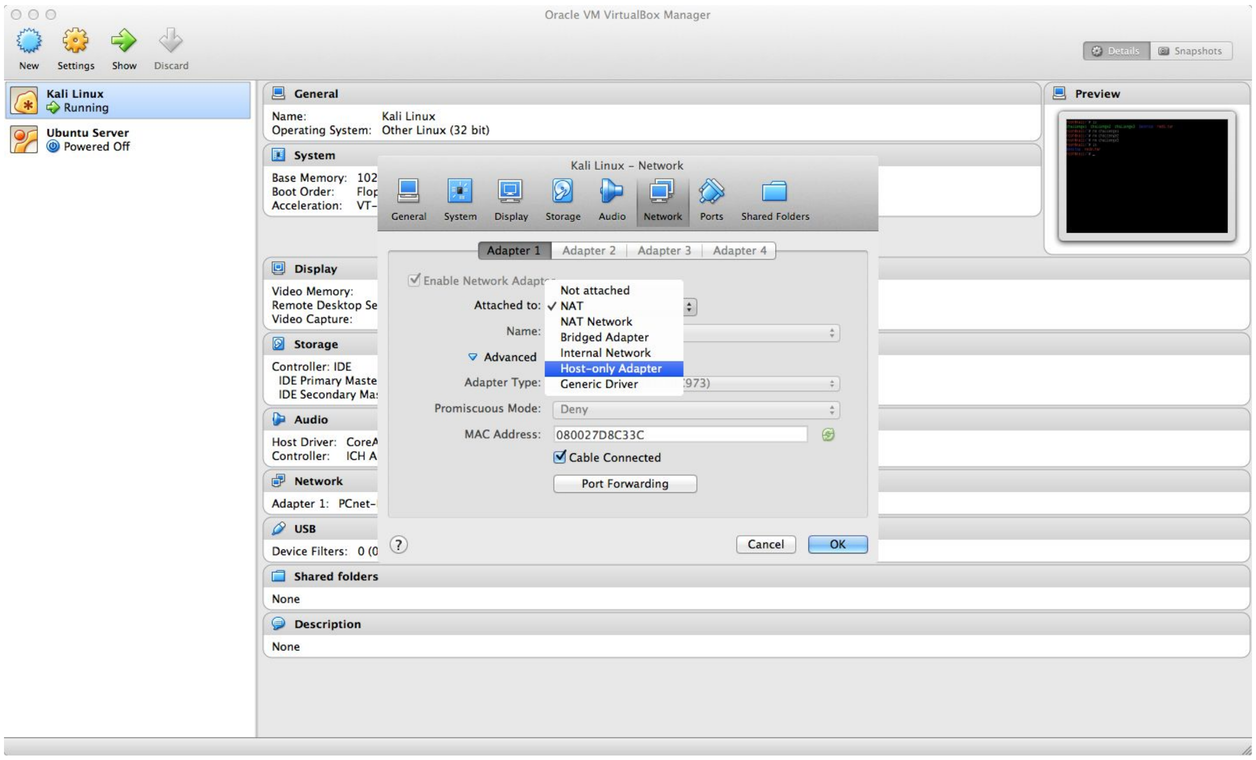
Figure 11: Why hello there

Now that we know now Netcat and Wireshark works, let’s combine them together to take a look at TCP segments!

Changing Network Adapter to Host-only

In order to follow the rest of the document, you'll need to change the VM's network adapter to Host-only. What this will do is create a network that is completely contained within the host computer (as the name implies.) This network will be isolated from the general internet and will allow Wireshark to only capture the packets that the VMs are exchanging making our captures much easier to analyze. (Note: your VMs will not be able to access the Internet using a Host-only adapter.)

If you're using VirtualBox, you can change your network adapter to Host-only by going to the VM's settings, clicking on the Network tab, and selecting Host-only from the dropdown for the "Attached to" option (Figure 12).



Name: Kali Linux
Operating System: Other Linux (32 bit)

Attached to: ✓ NAT
Name:
Advanced
Adapter Type:
Promiscuous Mode: Deny
MAC Address: 080027D8C33C
✓ Cable Connected
Port Forwarding

Figure 12: Changing to Host-only Adapter

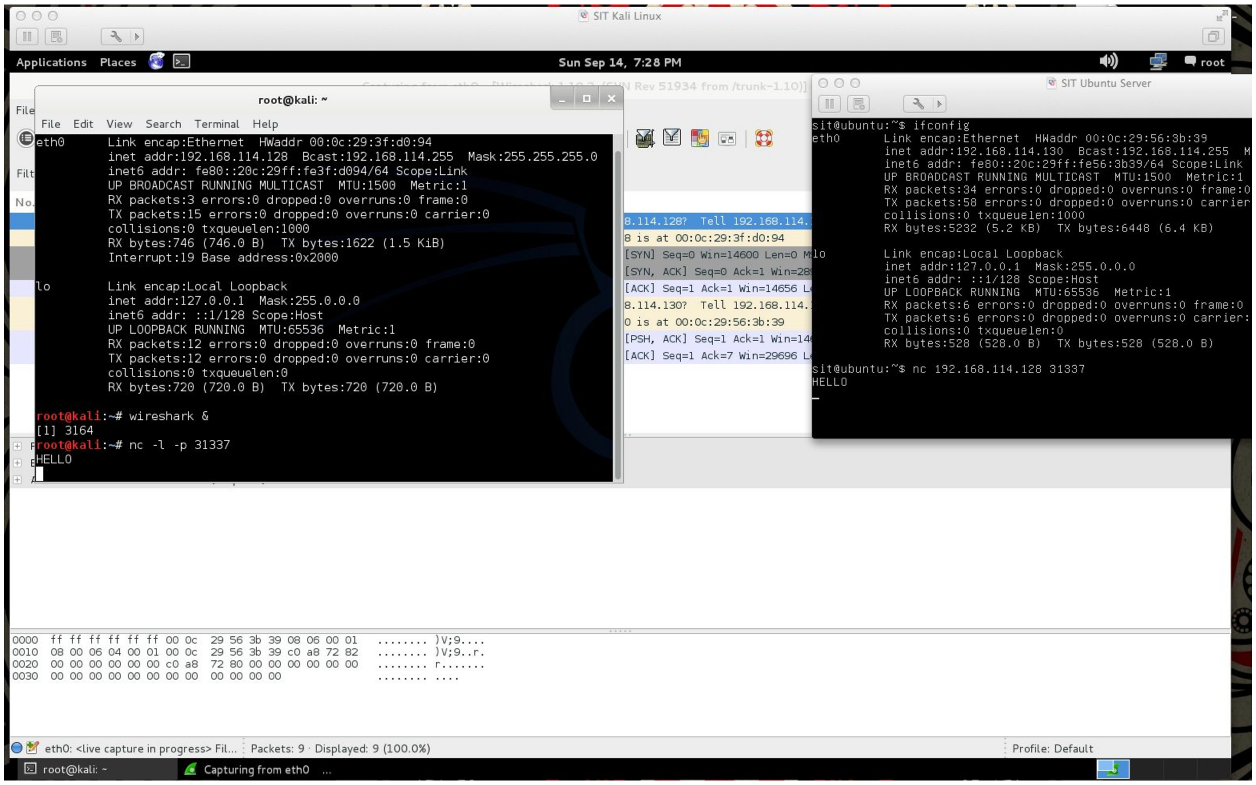
If your VM needs to access the Internet, I would recommend switching back to the NAT adapter.

Wireshark

Now that we have a general idea of how TCP/IP packets are structured, let's use Netcat to send some information from one VM to another while running a Wireshark capture.

Like before, let's start up Wireshark on Kali Linux by using `*wireshark &*`and find your ip address by using `*ifconfig*`. Once you have your ip address, create an Netcat listener on port 31337 by using `*nc -l -p 31337*`. What this listener will do is essentially "listen" for a client to connect on port 31337. And, using Ubuntu Server, we'll connect to our Netcat listener by using `*nc <kali-ip-address> 31337*`.

On your Ubuntu Server VM, you'll see that now you're on a blank newline which means that you're connected! Go ahead and type HELLO and press enter. You should see HELLO appear on your Kali Linux terminal too (Figure 13).



```
eth0      Link encap:Ethernet  HWaddr 00:0c:29:3f:d0:94
          inet addr:192.168.114.128  Bcast:192.168.114.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe3f:d094/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:746 (746.0 B)  TX bytes:1622 (1.5 KiB)
          Interrupt:19 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:12 errors:0 dropped:0 overruns:0 frame:0
          TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:720 (720.0 B)  TX bytes:720 (720.0 B)

root@kali:~# wireshark &
[1] 3164
root@kali:~# nc -l -p 31337
HELLO
```

```
8.114.128?  Tell 192.168.114.
8 is at 00:0c:29:3f:d0:94
[SYN] Seq=0 Win=14600 Len=0 M
[SYN, ACK] Seq=0 Ack=1 Win=28
[ACK] Seq=1 Ack=1 Win=14656 L
8.114.130?  Tell 192.168.114.
0 is at 00:0c:29:56:3b:39
[PSH, ACK] Seq=1 Ack=1 Win=14
[ACK] Seq=1 Ack=7 Win=29696 L
```

```
sit@ubuntu:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:56:3b:39
          inet addr:192.168.114.130  Bcast:192.168.114.255  M
          inet6 addr: fe80::20c:29ff:fe56:3b39/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:34 errors:0 dropped:0 overruns:0 frame:0
          TX packets:58 errors:0 dropped:0 overruns:0 carrier
          collisions:0 txqueuelen:1000
          RX bytes:5232 (5.2 KB)  TX bytes:6448 (6.4 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:
          collisions:0 txqueuelen:0
          RX bytes:528 (528.0 B)  TX bytes:528 (528.0 B)

sit@ubuntu:~$ nc 192.168.114.128 31337
HELLO
_
```

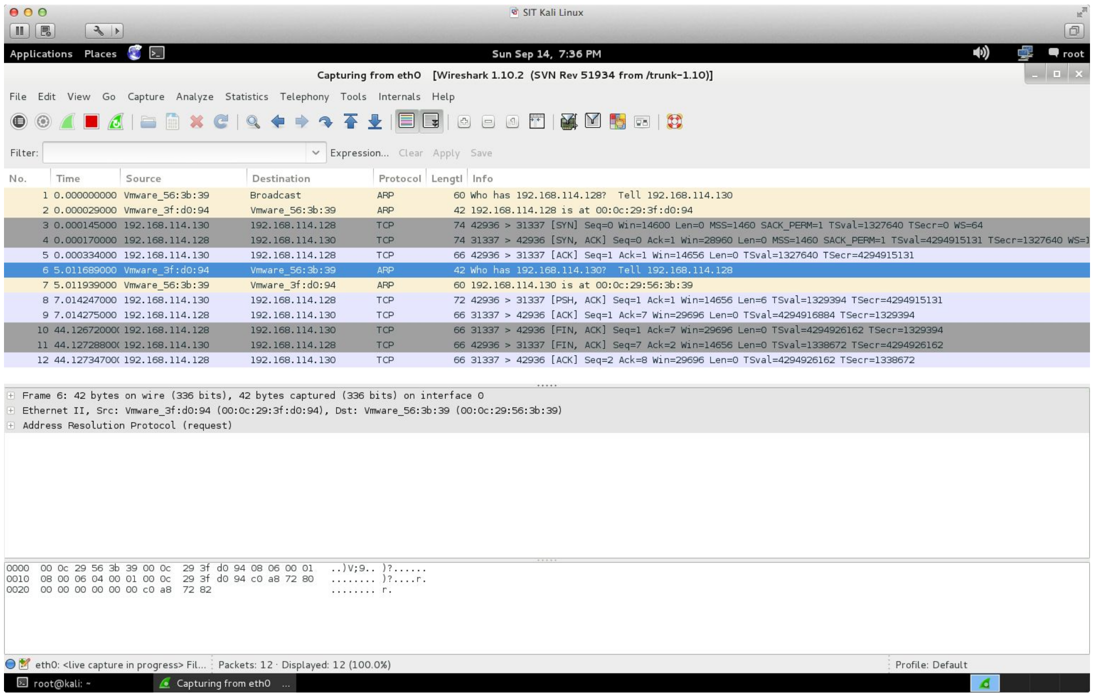
```
0000  ff ff ff ff ff ff 00 0c  29 56 3b 39 08 06 00 01   ........ )V;9....
0010  08 00 06 04 00 01 00 0c  29 56 3b 39 c0 a8 72 82   ........ )V;9..r.
0020  00 00 00 00 00 00 c0 a8  72 80 00 00 00 00 00 00   ........ r.......
0030  00 00 00 00 00 00 00 00  00 00 00 00               ........ ....
```

eth0: <live capture in progress> Fil... Packets: 9 · Displayed: 9 (100.0%) Profile: Default

root@kali: ~ Capturing from eth0 ...

Figure 13: It worked!

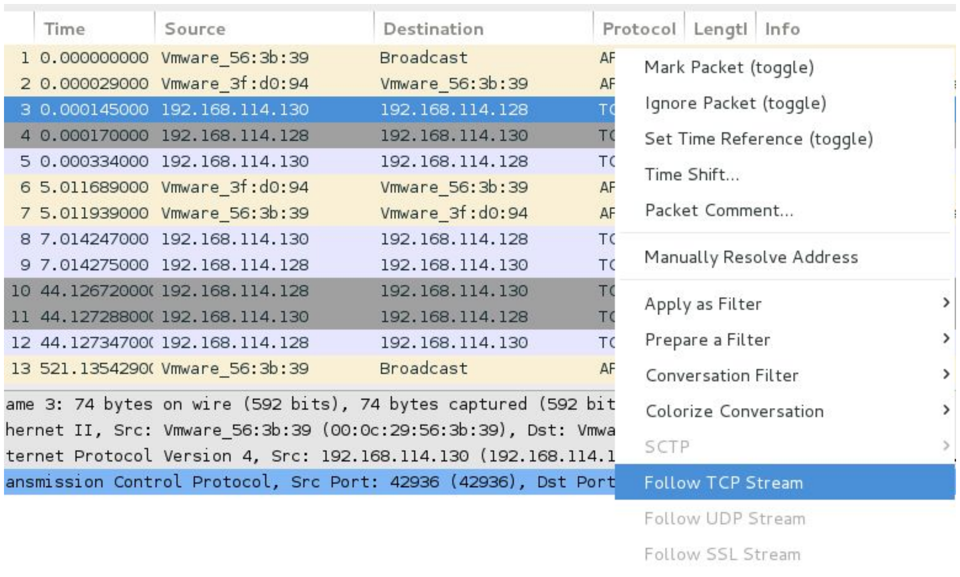
If you see HELLO on both your Kali Linux and Ubuntu Server shells, it worked! And, we finally have some packets to analyze in Wireshark (Figure 14).



```
No.  Time         Source             Destination        Protocol Lengtl Info
   1 0.000000000  Vmware_56:3b:39    Broadcast          ARP        60 Who has 192.168.114.128?  Tell 192.168.114.130
   2 0.000029000  Vmware_3f:d0:94    Vmware_56:3b:39    ARP        42 192.168.114.128 is at 00:0c:29:3f:d0:94
   3 0.000145000  192.168.114.130    192.168.114.128    TCP        74 42936 > 31337 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=1327640 TSecr=0 WS=64
   4 0.000170000  192.168.114.128    192.168.114.130    TCP        74 31337 > 42936 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=4294915131 TSecr=1327640 WS=1
   5 0.000334000  192.168.114.130    192.168.114.128    TCP        66 42936 > 31337 [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=1327640 TSecr=4294915131
   6 5.011689000  Vmware_3f:d0:94    Vmware_56:3b:39    ARP        42 Who has 192.168.114.130?  Tell 192.168.114.128
   7 5.011939000  Vmware_56:3b:39    Vmware_3f:d0:94    ARP        60 192.168.114.130 is at 00:0c:29:56:3b:39
   8 7.014247000  192.168.114.130    192.168.114.128    TCP        72 42936 > 31337 [PSH, ACK] Seq=1 Ack=1 Win=14656 Len=6 TSval=1329394 TSecr=4294915131
   9 7.014275000  192.168.114.128    192.168.114.130    TCP        66 31337 > 42936 [ACK] Seq=1 Ack=7 Win=29696 Len=0 TSval=4294916884 TSecr=1329394
  10 44.12672000( 192.168.114.128    192.168.114.130    TCP        66 31337 > 42936 [FIN, ACK] Seq=1 Ack=7 Win=29696 Len=0 TSval=4294926162 TSecr=1329394
  11 44.12728800( 192.168.114.130    192.168.114.128    TCP        66 42936 > 31337 [FIN, ACK] Seq=7 Ack=2 Win=14656 Len=0 TSval=1338672 TSecr=4294926162
  12 44.12734700( 192.168.114.128    192.168.114.130    TCP        66 31337 > 42936 [ACK] Seq=2 Ack=8 Win=29696 Len=0 TSval=4294926162 TSecr=1338672

⊞ Frame 6: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
⊞ Ethernet II, Src: Vmware_3f:d0:94 (00:0c:29:3f:d0:94), Dst: Vmware_56:3b:39 (00:0c:29:56:3b:39)
⊞ Address Resolution Protocol (request)

0000  00 0c 29 56 3b 39 00 0c  29 3f d0 94 08 06 00 01   ..)V;9.. )?......
0010  08 00 06 04 00 01 00 0c  29 3f d0 94 c0 a8 72 80   ........ )?....r.
0020  00 00 00 00 00 00 c0 a8  72 82                     ........ r.

eth0: <live capture in progress> Fil...   Packets: 12 · Displayed: 12 (100.0%)
root@kali: ~    Capturing from eth0 ...
```

Figure 14: Packets!

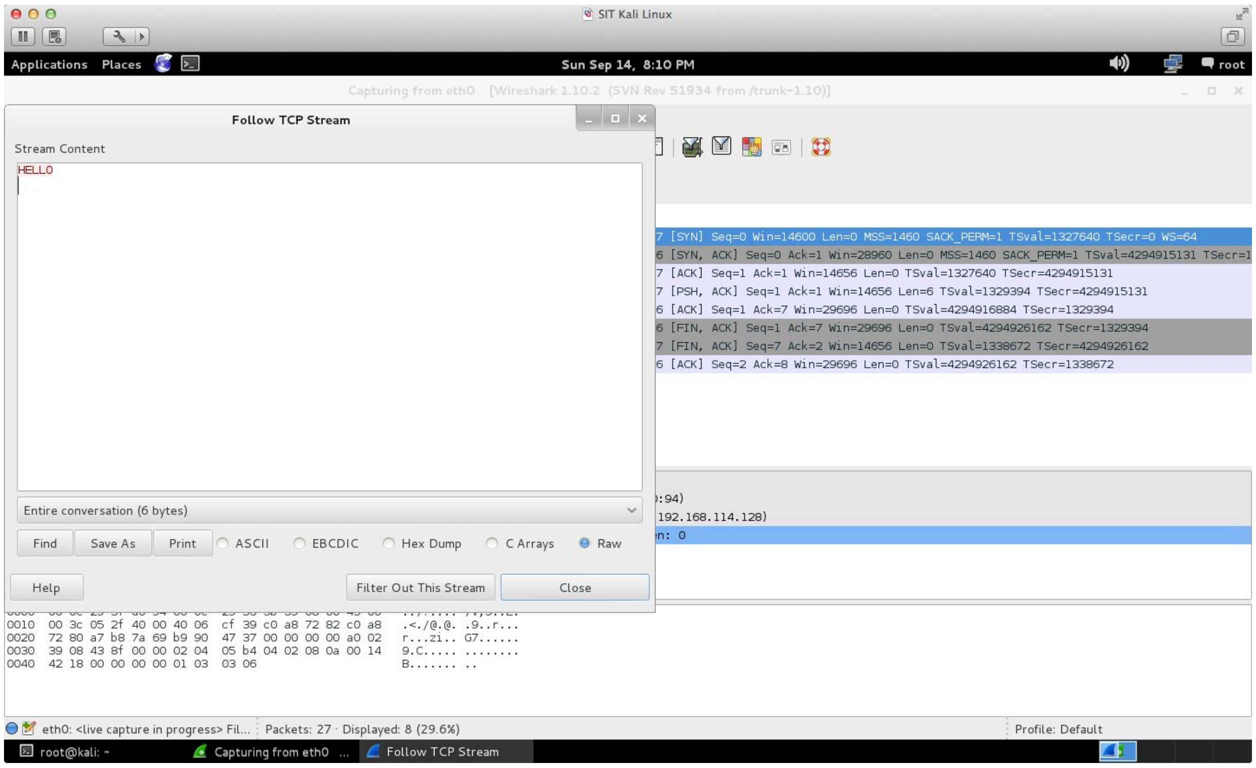
Looking at our capture list, we have a few ARP packets that don't belong to our Netcat demo. To filter these out, let's go ahead and find the first packet involved in our Netcat demo. The best way to find a packet involved would be to look for the first TCP packet that has a source IP address of our Ubuntu Server VM and the destination IP address of our Kali Linux VM and right click on the line and select "Follow TCP Stream" (Figure 15).



```
    Time          Source            Destination       Protocol  Lengtl  Info
 1  0.000000000   Vmware_56:3b:39   Broadcast         AF
 2  0.000029000   Vmware_3f:d0:94   Vmware_56:3b:39   AF
 3  0.000145000   192.168.114.130   192.168.114.128   TC
 4  0.000170000   192.168.114.128   192.168.114.130   TC
 5  0.000334000   192.168.114.130   192.168.114.128   TC
 6  5.011689000   Vmware_3f:d0:94   Vmware_56:3b:39   AF
 7  5.011939000   Vmware_56:3b:39   Vmware_3f:d0:94   AF
 8  7.014247000   192.168.114.130   192.168.114.128   TC
 9  7.014275000   192.168.114.128   192.168.114.130   TC
10  44.12672000(  192.168.114.128   192.168.114.130   TC
11  44.12728800(  192.168.114.130   192.168.114.128   TC
12  44.12734700(  192.168.114.128   192.168.114.130   TC
13  521.1354290(  Vmware_56:3b:39   Broadcast         AF

ame 3: 74 bytes on wire (592 bits), 74 bytes captured (592 bit
hernet II, Src: Vmware_56:3b:39 (00:0c:29:56:3b:39), Dst: Vmwa
ternet Protocol Version 4, Src: 192.168.114.130 (192.168.114.1
ansmission Control Protocol, Src Port: 42936 (42936), Dst Port
```

Figure 15: Following a TCP Stream

You should see a window pop up with the TCP Stream Content (in this case, it should just be HELLO - Figure 16) and the packets in the Capture Window should be only those related to our Netcat demo.



```
HELLO
```

Entire conversation (6 bytes)

```
7 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=1327640 TSecr=0 WS=64
6 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=4294915131 TSecr=1
7 [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=1327640 TSecr=4294915131
7 [PSH, ACK] Seq=1 Ack=1 Win=14656 Len=6 TSval=1329394 TSecr=4294915131
6 [ACK] Seq=1 Ack=7 Win=29696 Len=0 TSval=4294916884 TSecr=1329394
6 [FIN, ACK] Seq=1 Ack=7 Win=29696 Len=0 TSval=4294926162 TSecr=1329394
7 [FIN, ACK] Seq=7 Ack=2 Win=14656 Len=0 TSval=1338672 TSecr=4294926162
6 [ACK] Seq=2 Ack=8 Win=29696 Len=0 TSval=4294926162 TSecr=1338672
```

```
):94)
192.168.114.128)
n: 0
```

```
0010  00 3c 05 2f 40 00 40 06  cf 39 c0 a8 72 82 c0 a8   .</@.@. .9..r...
0020  72 80 a7 b8 7a 69 b9 90  47 37 00 00 00 00 a0 02   r...zi.. G7......
0030  39 08 43 8f 00 00 02 04  05 b4 04 02 08 0a 00 14   9.C..... ........
0040  42 18 00 00 00 00 01 03  03 06                     B....... ..
```

eth0: <live capture in progress> Fil... Packets: 27 · Displayed: 8 (29.6%)

Figure 16: Our Netcat demo's TCP stream

One question that we should answer here is how exactly does Wireshark determine our Stream Content? Well, recall back to our TCP packet structure. The information from the data section of the TCP packets from our TCP stream is what Wireshark displays in the Stream Content. Pretty neat, huh?

Exercises

There will be three exercises to go along with this module. All three exercises are designed to showcase different features of Wireshark and may or may not get harder with difficulty. The exercises will be available at the UFSIT meeting on Wednesday September 17, 2014 from 1900-2100 in CSE E309.

Conclusion

While we covered a lot of content in this module, we only scratched the surface in terms of networking, Netcat, and Wireshark. Netcat is a really really cool utility (check out "Netcat Without Netcat" from HakTip) and Wireshark has a ton of cool features that we didn't even discuss, but now you know enough about the basics to start doing cool things with Netcat and Wireshark! If you're hungry for more Wireshark, check out sample pcaps available for download here: http://wiki.wireshark.org/SampleCaptures.

Recommended Resources

If you're looking for more information on networking in general, Wikipedia hopping is probably the best way that you can learn more, so here are some links to get you started:

- https://en.wikipedia.org/wiki/OSI_model
- https://en.wikipedia.org/wiki/Transmission_Control_Protocol
- https://en.wikipedia.org/wiki/Internet_Protocol
- https://en.wikipedia.org/wiki/Internet_protocol_suite
- https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol

Netcat:

- Netcat Cheat Sheet by Ed Skoudis: https://www.sans.org/security-resources/sec560/netcat_cheat_sheet_v1.pdf
- How To Use Netcat to Establish and Test TCP and UDP Connections on a VPS by Justin Ellingwood: https://www.digitalocean.com/community/tutorials/how-to-use-netcat-to-establish-and-test-tcp-and-udp-connections-on-a-vps

Wireshark:

- Wireshark User's Guide: https://www.wireshark.org/docs/wsug_html_chunked/
- Wireshark Sample Captures: http://wiki.wireshark.org/SampleCaptures

This module was written by Andrew Kerr for the University of Florida Student InfoSec Team.