Network Troubleshooting Using Packet Capture Utilities – UUASC – 6/2/2005

1.0 Introduction

Purpose

The purpose of this paper is demonstrate how to monitor and troubleshoot common network based applications using standard UNIX utilities packet capture utilities. A basic understanding of the TCP/IP stack and Ethernet hardware is assumed.

Packet Capture/"Sniffing the Wire"

Promiscuous mode is generally used to refer to the practice of putting a network card into a setting so that it passes all traffic it receives to the CPU rather than just packets addressed to it. Many operating systems require superuser privileges to operate in promiscuous mode. A **non**-routing node in promiscuous mode can generally only monitor traffic to and from other nodes within the same collision domain (for Ethernet and Wireless LAN) or ring (for Token Ring or FDDI), which is why network switches are used to combat malicious use of promiscuous mode. A router may monitor all traffic that it routes. Promiscuous mode is commonly used to diagnose network connectivity issues. Some programs like Ethereal, tcpdump, and AirSnort (for wireless LANs) make use of this feature to show the user all the data being transferred over the network. Some programs like FTP and Telnet transfer data and passwords in clear text, without encryption, and network scanners can see this data. Therefore, computer users are encouraged to stay away from programs like telnet and use more secure ones such as SSH.

2.0 libpcap Based Tools

The **libpcap** library is a system-independent interface for user-level packet capture. Many UNIX utilities use the **libpcap** interface as their underlying packet capture engine. Due to the portability of this code, all utilities that use the **libpcap** library share the same syntax. The most common utilities that use the **libpcap** library are **tcpdump** and **ethereal**.

2.1 The tcpdump Utility

The **tcpdump** utility is the most common packet capture utility for UNIX based systems. It is highly versatile and filterable. Here is a standard run of the command:

```
box:~# tcpdump
tcpdump: WARNING: eth0: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
```

There are a few very helpful options out there. In the following example, the "**-i**" specifies another Ethernet interface besides the default (**eth0**). Also included is the "**-n**" option which turns of host and port resolution. The **tcpdump** utility will attempt to resolve IP address, which can lead to significant delays in output due to the latency of network lookups. This can be a very handy option when attempting to monitor high volumes of traffic.

```
box:~# tcpdump -ni eth1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
19:47:22.935554 IP 192.168.1.105.32783 > 67.110.253.165.993: P 3432387228:3432387265(37)
ack 2742259796 win 63712 <nop,nop,timestamp 239675 1064682926>
19:47:22.967508 IP 67.110.253.165.993 > 192.168.1.105.32783: P 1:54(53) ack 37 win 1984
<nop,nop,timestamp 1064879093 239675>
```

Here is a breakdown of a single packet:

Real Time:	`19:47:22.967508`
Source IP Address:	`IP 67.110.253.165.993`
Direction of Packet Flow:	`>`
Destination Address:	`192.168.1.105.32783:`
TCP Flags:	`P`
TCP Source SYN Number:	`1:`
Next TCP SYN Number	`54(53)` # original SYN (1) + payload (53) = next SYN (54)
TCP ACK NUMBER:	`ack 37`
TCP Window Size:	`win 1984`
TCP Options:	`<nop,nop,timestamp 1064879093 239675>`

If the volume of traffic to be monitored is too great for a standard terminal window buffer, the captured packets can be written to a file instead of STDOUT with a "**`-w`**" (write) option and then read back in with a "**`-r`**" (read) option.

```
box:~# tcpdump -w /tmp/tcp.out -ni eth1
tcpdump: listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
46 packets captured
46 packets received by filter
0 packets dropped by kernel

box:~# tcpdump -r /tmp/tcp.out -ni eth1
reading from file /tmp/tcp.out, link-type EN10MB (Ethernet)
19:56:07.190888 IP 192.168.1.105.32783 > 67.110.253.165.993: P 3432387731:3432387768(37)
ack 2742260475 win 63712 <nop,nop,timestamp 292100 1065283060>
19:56:07.227315 IP 67.110.253.165.993 > 192.168.1.105.32783: P 1:54(53) ack 37 win 1984
<nop,nop,timestamp 1065403449 292100>
<<snip>>
```

2.2 The `ethereal` Utility

Just like **`tcpdump`**, **`ethereal`** is based on the **`libpcap`** interface. There are two main versions of **`ethereal`**. There is the text version called "**`tethereal`**" and the GUI based version called "**`ethereal`**". The text based version is very similar in syntax to the **`tcpdump`** command syntax. Once again, this is because they use the same underlying **libpcap** engine.

```
box:~# tethereal -w /tmp/ethereal.out -ni eth1
Capturing on eth1
0.327450 192.168.1.105 -> 67.110.253.165 TLS Application Data
0.361175 67.110.253.165 -> 192.168.1.105 TLS Application Data
0.361220 192.168.1.105 -> 67.110.253.165 TCP 32783 > 993 [ACK] Seq=37 Ack=53 Win=63712
Len=0 TSV=389797 TSER=1066380554
0.363460 192.168.1.105 -> 67.110.253.165 TLS Application Data
0.410951 67.110.253.165 -> 192.168.1.105 TLS Application Data

box:~# tethereal -r /tmp/ethereal.out
6   2.543822 192.168.1.105 -> 67.110.253.165 TLS Application Data
7   2.593330 67.110.253.165 -> 192.168.1.105 TLS Application Data
8   2.593375 192.168.1.105 -> 67.110.253.165 TCP 32783 > imaps [ACK] Seq=37 Ack=53
Win=63712 Len=0 TSV=412045 TSER=1066603077
9   2.595989 192.168.1.105 -> 67.110.253.165 TLS Application Data
```

2.3 The dsniff Utility

Unlike the previously mentioned utilities, **dsniff** takes packet capture one level further. Using the underlying **libpcap** engine, **dsniff** takes the packets captured and attempts to report something a little more useful. The **dsniff** program is one of many utilities in the **dsniff** package. The standard **dsniff** command will attempt to capture and replay all unencrypted sessions including: **FTP**, **telnet**, **SMTP**, **IMAP**, and **POP**. The following example demonstrates how to use **dsniff** to audit **telnet** and **ftp** sessions:

```
box:~# dsniff -ni eth1
dsniff: listening on eth1
-----------------
06/01/05 20:35:46 tcp 192.168.1.105.32883 -> 192.168.1.220.21 (ftp)
USER darren
PASS darren$$$$

-----------------
06/01/05 20:37:53 tcp 192.168.1.105.32889 -> 192.168.1.220.23 (telnet)
darren
darren$$$$
ls
ls -l
ps -ef
exit
```

2.4 The snort Utility

The **snort** utility is the most common open source intrusion detection system. Like **dsniff**, it attempts to make sense out of traffic. Whereas simply reports clear text payloads, **snort** attempts to identify malicious traffic patterns using signatures of known malicious packets. Like all the other utilities, **snort** uses **libpcap** as the underlying engine. The following is a very basic run of **snort**.

```
box:~# snort -i eth1-D -d  -u snort -g snort -c /etc/snort/snort.conf
```

The **snort** utility will run in daemon mode in the background. All alerts are written to a text file **/var/log/snort/alert**. Here is a sample scan from a possible intruder:

```
[root@targus ~]# nmap -p 22 -sX 192.168.1.105
```

This questionable port scan of the ssh port (22) is obfuscated by the "**-sX**" (Christmas scan) switch.

Here is the output from the alert file on the host running **snort**:

```
box:~# tail -f /var/log/snort/alert

[**] [1:1228:7] SCAN nmap XMAS [**]
[Classification: Attempted Information Leak] [Priority: 2]
06/01-20:42:02.813099 192.168.1.220:37325 -> 192.168.1.105:22
TCP TTL:47 TOS:0x0 ID:15194 IpLen:20 DgmLen:40
**U*P**F Seq: 0x82A00B2  Ack: 0x0  Win: 0x1000  TcpLen: 20  UrgPtr: 0x0
[Xref => http://www.whitehats.com/info/IDS30]
```

3.0 Sun Solaris **snoop** Utility

Snoop is specific to Sun Microsystem's Solaris UNIX. Although there is a port of **tcpdump** for Solaris, there is no port of **snoop** for LINUX. Much like **tcpdump**, it is a utility that puts your system's interface(s) in promiscuous mode. Although similar in design goals, since **snoop** uses it's own packet capture library, the options are a little different. The following example demonstrates how to run snoop on an alternate interface (**-d**) with name resolution disabled (**-r**).

```
pilate -> snoop  -r -d hme0
Using device /dev/hme (promiscuous mode)
66.27.208.74 -> 67.110.253.164 TCP D=22 S=32897 Ack=1172114638 Seq=1267082364 Len=0
Win=11200 Options=<nop,nop,tstamp 663391 727479722>
67.110.253.164 -> 66.27.208.74 TCP D=32897 S=22 Push Ack=1267082364 Seq=1172114638
Len=208 Win=47824 Options=<nop,nop,tstamp 727479725 663391>
```

Like **libpcap** utilities, **snoop** also enables the redirection of packets to a file instead of STDOUT.

```
pilate -> snoop -o /tmp/snoop.out -r -d hme0
Using device /dev/hme (promiscuous mode)
14

pilate -> snoop -i /tmp/snoop.out
  1   0.00000 cpe-66-27-208-74.socal.res.rr.com -> pilate       TCP D=22 S=32897
Ack=1172118318 Seq=1267084668 Len=0 Win=15600 Options=<nop,nop,tstamp 670581 727486912>
  2   0.00005       pilate -> cpe-66-27-208-74.socal.res.rr.com TCP D=32897 S=22 Push
Ack=1267084668 Seq=1172118318 Len=112 Win=47824 Options=<nop,nop,tstamp 727486915 670581>
```

4.0 Using Filter Expressions

It may be easy to identify specific traffic streams on small or idle networks. It will be much harder to accomplish this on large WAN or saturated networks. The ability to use filter expressions is extremely important in these cases to cut out unwanted “noise” packets from the traffic in question. Fortunately, both the **libpcap** based utilities and the **snoop** utility all use the same filter syntax. There are many ways to filter traffic in all utilities, the most common filters are by port, protocol, and host. The following example tracks only **telnet** traffic and host 192.168.1.105:

```
[root@targus ~]# tcpdump -ni eth0 port 23 and host 192.168.1.105
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
21:13:26.905262 IP 192.168.1.105.32899 > 192.168.1.220.telnet: S 1903904803:1903904803(0)
win 5840 <mss 1460,sackOK,timestamp 722613 0,nop,wscale 0>

[root@targus ~]# tethereal -ni eth0 port 23 and host 192.168.1.105
Capturing on eth0
  0.000000 192.168.1.105 -> 192.168.1.220 TCP 32900 > 23 [SYN] Seq=0 Ack=0 Win=5840 Len=0
MSS=1460 TSV=729689 TSER=0 WS=0

box:~# dsniff -ni eth1 port 23 and host 192.168.1.105
dsniff: listening on eth1 [port 23 and host 192.168.1.105]
-----------------
06/01/05 21:11:16 tcp 192.168.1.105.32901 -> 192.168.1.220.23 (telnet)
root
cangetin

pilate -> snoop -r -d hme0 port 53 and host 192.168.1.105
```

There are other cases where an administrator may want to capture all but certain types of traffic. Specifically, a lot of “noise” can be made if one is trying to run a packet capture while logged into the remote host. Much of the traffic generated will be the control traffic back to that host. The following example shows how to filter the **ssh** control traffic to and from the control connection (192.168.1.105 connected to 192.168.1.220 as **root**) and all DNS traffic.

```
[root@targus ~]# who
root     pts/2        Jun  1 21:30 (192.168.1.105)

[root@targus ~]# tcpdump -ni eth0 not host 192.168.1.105 and not port 53
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
21:32:07.692524 IP 216.93.214.50.51606 > 192.168.1.220.ssh: S 2550704294:2550704294(0)
win 5840 <mss 1460,sackOK,timestamp 431608120 0,nop,wscale 2>
21:32:07.692596 IP 192.168.1.220.ssh > 216.93.214.50.51606: S 729994889:729994889(0) ack
2550704295 win 5792 <mss 1460,sackOK,timestamp 111008380 431608120,nop,wscale 2>
21:32:07.796911 IP 216.93.214.50.51606 > 192.168.1.220.ssh: . ack 1 win 1460
<nop,nop,timestamp 431608221 111008380>
```

5.0 Protocol Layer Problems

5.1.0 ARP Layer Problems – No ARP Reply

The **ping** command is often used to test whether or not a remote host has a configured network stack. When a **ping** hangs or no reply is received, there is an assumption that the host is not up or working. This may be true, however, there could be multiple issues at the ARP layer preventing a working host from communicating on the network. The following example demonstrates what an expected ARP exchange looks like BEFORE a **ping** command can commence. It demonstrates the ARP REQUEST by the source host followed by the ARP REPLY from the destination host.

```
box:~# ping 192.168.1.102
PING 192.168.1.102 (192.168.1.102) 56(84) bytes of data.
64 bytes from 192.168.1.102: icmp_seq=1 ttl=128 time=5.83 ms

box:~# tcpdump -ni eth1 arp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
21:43:55.459856 arp who-has 192.168.1.102 tell 192.168.1.105
21:43:55.462638 arp reply 192.168.1.102 is-at 00:0f:1f:17:ab:a7
```

Here is an example of a failed **ping** attempt. Notice that the ARP REQUEST was never answered by the destination host (192.168.1.107). This could mean that the destination host is not online.

```
box:~# ping 192.168.1.107
PING 192.168.1.107 (192.168.1.107) 56(84) bytes of data.
From 192.168.1.105 icmp_seq=1 Destination Host Unreachable
From 192.168.1.105 icmp_seq=2 Destination Host Unreachable

box:~# tcpdump -ni eth1  arp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
21:36:48.995766 arp who-has 192.168.1.107 tell 192.168.1.105
21:36:49.992668 arp who-has 192.168.1.107 tell 192.168.1.105
21:36:50.992667 arp who-has 192.168.1.107 tell 192.168.1.105
```

5.1.1 ARP Layer Problems – Duplicate IP Addresses

When two hosts have the same IP Address assigned, there will be two ARP REPLY to the ARP REQUEST. The first reply enters the source host's ARP table. The problem is that the first reply may be the wrong host. The following example demonstrates how a Windows XP and LINUX host compete for the same IP address.

On the surface, the **ping** makes it appear that the destination host (expected to be LINUX) is up an responding.

```
box:~# ping 192.168.1.102
```

```
PING 192.168.1.102 (192.168.1.102) 56(84) bytes of data.
64 bytes from 192.168.1.102: icmp_seq=1 ttl=128 time=5.83 ms
```

However, a capture of ARP traffic shows that two replies were sent to the original request. The first reply was from the Windows host. The LINUX host's entry came after.

```
box:~# tcpdump -ni eth1 arp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
22:04:35.074216 arp who-has 192.168.1.102 tell 192.168.1.105
22:04:35.078448 arp reply 192.168.1.102 is-at 00:40:f4:83:48:24 # Windows Host
22:04:35.079562 arp reply 192.168.1.102 is-at 00:0f:1f:17:ab:a7 # LINUX Host
```

A check of the ARP cache shows that the Windows XP Ethernet address is the one populated in the cache.

```
box:~# arp -a
targus (192.168.1.220) at 00:02:55:74:41:1B [ether] on eth1
? (192.168.1.1) at 00:06:25:77:63:8B [ether] on eth1
? (192.168.1.102) at 00:40:f4:83:48:24 [ether] on eth1
```

An attempt to use `ssh` to connect to the remote host fails because the source host is attempting to connect to the Windows XP host instead of the LINUX host using the Windows XP Ethernet address.

```
box:~# ssh -v 192.168.1.102
OpenSSH_3.8.1p1 Debian-8.sarge.4, OpenSSL 0.9.7e 25 Oct 2004
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: Connecting to 192.168.1.102 [192.168.1.102] port 22.
^C
```

5.2.0 IP Problems – Misconfigured Broadcast Address

The broadcast address is often overlooked when determining network problems. Some network communications rely on a properly configured broadcast address including: NTP, RIPv1, and ICMP Broadcast pings. The following example demonstrates a standard ICMP broadcast `ping` an associated reply.

```
box:~# ifconfig eth1
eth1      Link encap:Ethernet  HWaddr 00:06:53:E4:8D:B8
          inet addr:192.168.1.105  Bcast:192.168.1.255  Mask:255.255.255.0

box:~# ping -b 192.168.1.255
WARNING: pinging broadcast address
PING 192.168.1.255 (192.168.1.255) 56(84) bytes of data.
64 bytes from 192.168.1.105: icmp_seq=1 ttl=64 time=0.052 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=150 time=3.54 ms (DUP!)
64 bytes from 192.168.1.220: icmp_seq=1 ttl=64 time=4.43 ms (DUP!)
```

If the broadcast address is not consistent with the rest of the hosts on the network, none of those hosts will reply to the broadcast `ping`. In the following example, the correct subnet mask 255.255.255.0. However, the source host has a misconfigured netmask of 255.0.0.0.

```
box:~# ifconfig eth1
eth1      Link encap:Ethernet  HWaddr 00:06:53:E4:8D:B8
          inet addr:192.168.1.105  Bcast:192.255.255.255  Mask:255.0.0.0
```

A broadcast **`ping`** returns no replies from the network.

```
box:~# ping -b 192.168.1.255
PING 192.168.1.255 (192.168.1.255) 56(84) bytes of data.
From 192.168.1.105 icmp_seq=1 Destination Host Unreachable
```

A packet capture further confirms the problem. The correct broadcast is 192.168.1.255, however since the source host's broadcast is 192.255.255.255, the source host is mistakingly trying to ARP for 192.168.1.255, thinking it is a real host. The source host will never receive a valid ARP reply as no host on the network can have a .255 in it's last octet.

```
box:~# tcpdump -ni eth1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
22:31:01.512750 arp who-has 192.168.1.255 tell 192.168.1.105
22:31:02.512665 arp who-has 192.168.1.255 tell 192.168.1.105
```

5.2.1 – IP Problems - Misconfigured Default Gateway

When a client host on a LAN can't communicate with the outside world, it can be one of 4 issues:

- no network connectivity
- misconfigured **`/etc/nsswitch.conf`**
- misconfigured or non-existent **`/etc/resolv.conf`** for DNS
- misconfigured or wrong gateway information

The first 3 issues can be solved by viewing files and checking physical links. There is no real way to tell if the gateway entry is truly routing packets. The following example demonstrates how to monitor whether the gateway is routing packets.

The client host is unable to reach a host on the Internet.

```
box:~# ping yahoo.com
ping: unknown host yahoo.com
```

The client has a gateway configured in the routing table. However, there is no way to tell whether the gateway is actually routing.

```
box:~# netstat -rn
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
192.168.1.0     0.0.0.0         255.255.255.0   U         0 0          0 eth1
0.0.0.0         192.168.1.220   0.0.0.0         UG        0 0          0 eth1
```

The following packet capture is taken from the router. The packet are coming from the source host of 192.168.1.105, but the interface is NOT showing the return packet.

```
[root@targus ~]# tcpdump -ni eth0 not port 22
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
09:01:39.063347 IP 192.168.1.105.32770 > 4.2.2.2.domain:  49762+ A? yahoo.com. (27)
09:01:44.075062 IP 192.168.1.105.32771 > 4.2.2.1.domain:  49762+ A? yahoo.com. (27)
```

From the client prospective, the router is on the network as it replies to a ping request.

```
box:~# ping 192.168.1.220
PING 192.168.1.220 (192.168.1.220) 56(84) bytes of data.
64 bytes from 192.168.1.220: icmp_seq=1 ttl=64 time=2.69 ms
64 bytes from 192.168.1.220: icmp_seq=2 ttl=64 time=2.95 ms
```

However, the client is not receiving any replies from it's DNS request. Packets are going to the router, however they are getting dropped.

```
box:~# tcpdump -ni eth1 not port 22
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
09:10:06.367238 IP 192.168.1.105.32772 > 4.2.2.2.53:  49099+ A? yahoo.com. (27)
09:10:11.381598 IP 192.168.1.105.32773 > 4.2.2.1.53:  49099+ A? yahoo.com. (27)
```

5.2.0 TCP Problems – Closed Ports

There are multiple reasons why a remote server may not respond to a client request. Examples have already been given to troubleshoot at the Ethernet, ARP, and IP levels. A common mistake is to assume that since a host is available at the IP level, it does not mean that it is available at the TCP level.

The host is available at the IP level as per the **`ping`** replies.

```
box:~# ping 192.168.1.220
PING 192.168.1.220 (192.168.1.220) 56(84) bytes of data.
64 bytes from 192.168.1.220: icmp_seq=1 ttl=64 time=2.71 ms
64 bytes from 192.168.1.220: icmp_seq=2 ttl=64 time=2.64 ms
```

The **`telnet`** service is not available to the client.

```
box:~# telnet 192.168.1.220
Trying 192.168.1.220...
telnet: Unable to connect to remote host: Connection refused
```

Taking a look at the packet capture, it is clear that the **telnet** service is not running. Standard TCP replies to closed ports is to send a "reset" flag to the source host.

```
box:~# tcpdump -ni eth1 port 23
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
09:18:12.495679 IP 192.168.1.105.32780 > 192.168.1.220.23: S 2557844136:2557844136(0) win
5840 <mss 1460,sackOK,timestamp 1643931 0,nop,wscale 0>
09:18:12.498346 IP 192.168.1.220.23 > 192.168.1.105.32780: R 0:0(0) ack 2557844137 win 0
```

5.2.1 TCP Problems – TCP Wrapped Services

TCP Wrappers have been around for quite some time. Their purpose is to do host based access control. Unlike closed ports, the port to the server is open. Upon connection to the server, a check is made to the **/etc/hosts.allow** and **/etc/hosts.deny** files. If the client is allowed to connect, then a standard TCP connection is made. If not, the server sends a TCP reset to the client. The following is an example of a TCP wrapped **ssh** service.

```
box:~# ssh 192.168.1.220
ssh_exchange_identification: Connection closed by remote host
```

A look at the packet capture shows that upon determining that the client is denied, the **ssh** server initiates a standard port closing through a series of FIN packet exchanges.

```
box:~# tcpdump -ni eth1 port 22
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
10:24:44.508208 IP 192.168.1.105.32786 > 192.168.1.220.22: S 2304868053:2304868053(0) win
5840 <mss 1460,sackOK,timestamp 2043132 0,nop,wscale 0>
10:24:44.510770 IP 192.168.1.220.22 > 192.168.1.105.32786: S 762146323:762146323(0) ack
2304868054 win 5792 <mss 1460,sackOK,timestamp 44455276 2043132,nop,wscale 2>
10:24:44.510807 IP 192.168.1.105.32786 > 192.168.1.220.22: . ack 1 win 5840
<nop,nop,timestamp 2043132 44455276>
10:24:49.526296 IP 192.168.1.220.22 > 192.168.1.105.32786: F 1:1(0) ack 1 win 1448
<nop,nop,timestamp 44460292 2043132>
10:24:49.526647 IP 192.168.1.105.32786 > 192.168.1.220.22: F 1:1(0) ack 2 win 5840
<nop,nop,timestamp 2043634 44460292>
10:24:49.529124 IP 192.168.1.220.22 > 192.168.1.105.32786: . ack 2 win 1448
<nop,nop,timestamp 44460295 2043634>
```

5.2.2 TCP Problems – Packet Filtered TCP Ports

When a port is blocked by a packet filter (IPTables or IPFilter for example), it may be open but filtered at the IP level. In this case, the client will send multiple SYN packets to the server and the server will not respond simply because the packet has been dropped by the filter.

```
box:~# telnet 192.168.1.220
Trying 192.168.1.220...
telnet: Unable to connect to remote host: No route to host
```

A look at the packet capture shows that the client sent 3 TCP SYN packets to the `telnet` port (23) that were simply dropped by the server and not replied to in the client.

```
box:~# tcpdump -ni eth1 port 23
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
10:35:02.302120 IP 192.168.1.105.32787 > 192.168.1.220.23: S 2934062463:2934062463(0) win
5840 <mss 1460,sackOK,timestamp 2104912 0,nop,wscale 0>
10:35:13.847729 IP 192.168.1.105.32788 > 192.168.1.220.23: S 2951328215:2951328215(0) win
5840 <mss 1460,sackOK,timestamp 2106066 0,nop,wscale 0>
10:35:19.518837 IP 192.168.1.105.32789 > 192.168.1.220.23: S 2947681121:2947681121(0) win
5840 <mss 1460,sackOK,timestamp 2106633 0,nop,wscale 0>
```

6.0 Application Layer Problems

6.1.0 DHCP Problems – IP Address already assigned

The DHCP protocol is a largely transparent protocol on the network. The following packet capture shows the pieces that comprise a DHCP exchange between a client and a DHCP server.

```
[root@targus dhcp]# tethereal -ni eth1 not port 22
4.471802      0.0.0.0 -> 255.255.255.255 DHCP DHCP Discover - Transaction ID 0x803fcf50
```

After the initial `DHCP DISCOVER` made by the client, the server sends out an ARP request to the network to see if any other host has taken the IP Address from the DHCP pool.

```
4.475738 00:06:25:77:63:8b -> ff:ff:ff:ff:ff:ff ARP Who has 192.168.1.104?  Tell
192.168.1.1
```

If there are no ARP replies, then the DHCP server offers the IP address to the client. The client accepts the lease and the DHCP server acknowledges that the lease is now assigned.

```
5.135171  192.168.1.1 -> 255.255.255.255 DHCP DHCP Offer    - Transaction ID 0x803fcf50
5.135471      0.0.0.0 -> 255.255.255.255 DHCP DHCP Request  - Transaction ID 0x803fcf50
5.139041  192.168.1.1 -> 255.255.255.255 DHCP DHCP ACK      - Transaction ID 0x803fcf50
```

From this point on, a client will continue to ask for whatever address it previously had. If the client disconnects from the network and then tries to reconnect, the old IP address could easily be taken by another client. In the following packet capture, the client requests it's old IP address of `192.168.1.104`.

```
[root@targus dhcp]# tethereal -ni eth1 not port 22
Capturing on eth1
```

```
4.462923      0.0.0.0 -> 255.255.255.255 DHCP DHCP Request  - Transaction ID 0x6ccda4f
```

The DHCP server sent out an ARP request and received a valid ARP reply from another host.

```
4.465737 00:06:25:77:63:8b -> ff:ff:ff:ff:ff:ff ARP Who has 192.168.1.104?  Tell
192.168.1.1
4.465762 00:c3:61:f9:42:a7 -> 00:06:25:77:63:8b ARP 192.168.1.104 is at  00:c3:61:f9:42:a7
4.467191  192.168.1.1 -> 192.168.1.104 ICMP [Malformed Packet]
```

Therefore, the old requested address of **`192.168.1.104`** is no longer available and the DHCP server sends a "**`NAK`**" (no acknowledgment) packet back to the client, causing the client to start at the beginning with a **`DHCP DISCOVER.`**

```
4.471480  192.168.1.1 -> 255.255.255.255 DHCP DHCP NAK      - Transaction ID 0x6ccda4f
```

6.1.1 – DHCP Problems – Using DHCP for Manual Network Configuration

If there are hardware or software version issues that prevent a client from negotiating a lease from a DHCP server, then a NIC card can be configured manually with all the networking information from a **`DHCP ACK`** message from a DHCP server. The following packet capture displays verbose output of a **`DHCP ACK`**. This packet includes all information needed to successfully configure a NIC manually.

```
[root@targus ~]# ifconfig eth1 up
[root@targus ~]# ifup eth1
[root@targus ~]# tethereal -nVi eth1 port bootpc
Capturing on eth1

<<snip>>

    Your (client) IP address: 192.168.1.100 (192.168.1.100)
    Next server IP address: 192.168.1.1 (192.168.1.1)
    Relay agent IP address: 0.0.0.0 (0.0.0.0)
    Client MAC address: 00:40:f4:83:48:24 (CameoCom_83:48:24)
    Server host name not given
    Boot file name not given
    Magic cookie: (OK)
    Option 53: DHCP Message Type = DHCP ACK
    Option 1: Subnet Mask = 255.255.255.0
    Option 3: Router = 192.168.1.1
    Option 6: Domain Name Server
        IP Address: 66.75.160.62
        IP Address: 66.75.160.41
        IP Address: 66.75.160.37
    Option 15: Domain Name = "socal.rr.com"
    Option 51: IP Address Lease Time = 1 day
    Option 54: Server Identifier = 192.168.1.1
    End Option
    Padding
```

6.1.2 Samba Problems – Unable to Connect to Samba Server

The Samba protocol, often used to network Windows clients to UNIX servers, is largely a broadcast protocol. When failing to connect a Windows based client to a UNIX server, there is no useful debugging information available on the Windows host. There is an error message that simply states the "Network Path Not Found" or something similar. The following packet capture shows that the client and server can't communicate over Samba due to the differences in broadcast addresses. The client (192.168.1.102) is attempting to discover the server (192.168.1.220) , however, both have different broadcast domains and therefore can't here each other's requests.

```
[root@targus tmp]# tcpdump -ni eth0 not port 22 and not host 192.168.1.1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
13:51:58.319893 IP 192.168.1.102.netbios-ns > 192.168.1.255.netbios-ns: NBT UDP PACKET
(137): QUERY; REQUEST; BROADCAST
13:51:59.069690 IP 192.168.1.102.netbios-ns > 192.168.1.255.netbios-ns: NBT UDP PACKET
(137): QUERY; REQUEST; BROADCAST
13:51:59.820093 IP 192.168.1.102.netbios-ns > 192.168.1.255.netbios-ns: NBT UDP PACKET
(137): QUERY; REQUEST; BROADCAST
13:52:02.114031 IP 192.168.1.220.netbios-ns > 192.168.255.255.netbios-ns: NBT UDP PACKET
(137): QUERY; REQUEST; BROADCAST
13:52:04.113961 IP 192.168.1.220.netbios-ns > 192.168.255.255.netbios-ns: NBT UDP PACKET
(137): QUERY; REQUEST; BROADCAST
```

References

The dsniff project - http://www.monkey.org/~dugsong/dsniff/
The snoop utility - http://www.spitzner.net/snoop.html
The tcpdump/libpcap project - http://www.tcpdump.org/
The snort project – http://www.snort.org
The ethereal project - http://www.ethereal.com/
DHCP Protocol - http://www.faqs.org/rfcs/rfc1541.html
SMB Protocol - http://www.faqs.org/rfcs/rfc1001.html