Advanced Unix System Administration
Spring 2008
Homework 3 Solutions

1. *Networking on paper.* Here's an exercise to test your understanding of TCP/IP over Ethernet.

   a. Your company needs five different networks, four small networks of about 20 servers each, and a larger network of clients with addresses assigned by DHCP. You have the IP address range `172.17.42.0/24` to work with (I know this is RFC 1918 space – it's an example). Suggest a way to divide up this netblock into the networks you need.

      One solution: use `172.17.42.0/27`, `172.17.42.32/27`, `172.17.42.64/27`, and `172.17.42.96/27` for the servers and `172.17.42.128/25` for the clients. Each of the server networks will have 30 usable addresses (27 bits for the network address leaves 5 bits for the host address, which is 32 possibilities; the low value is used to identify the network, whereas the high value is the broadcast address), whereas the client network will have 126 addresses.

   b. A computer on an Ethernet network with MAC address `FF:FF:FE:09:42:A3` and IP address `172.17.42.37` sends the message `Hello, world!\r\n` via UDP from port 51500 to a computer with MAC address `FF:FF:FB:3D:28:9C` and IP address `172.17.42.58` on port 9. Describe each of the Ethernet frames resulting from this conversation. Assume the sender's ARP cache is empty at the beginning of the conversation.

      This is a short conversation – three packets. First, the ARP broadcast looking for `172.17.42.58`:

```
Ethernet II frame
Destination: FF:FF:FF:FF:FF:FF (broadcast)
Source: FF:FF:FE:09:42:A3
Type: 0x0806 (ARP)
ARP packet
Hardware type: 0x0001 (Ethernet)
Protocol: 0x0800 (IP)
Opcode: 0x0001 (request)
Sender MAC: FF:FF:FE:09:42:A3
Sender IP: 172.17.42.37
Target MAC: 00:00:00:00:00:00
Target IP: 172.17.42.58
```

      The reply to the ARP request is not broadcast:

```
Ethernet II frame
```

```
Destination: FF:FF:FE:09:42:A3
Source: FF:FF:FB:3D:28:9C
Type: 0x0806 (ARP)
ARP packet
Hardware type: 0x0001 (Ethernet)
Protocol: 0x0800 (IP)
Opcode: 0x0002 (reply)
Sender MAC: FF:FF:FB:3D:28:9C
Sender IP: 172.17.42.58
Target MAC: FF:FF:FE:09:42:A3
Target IP: 172.17.42.37
```

With the MAC address of the destination in hand, the message is sent as a single UDP packet:

```
Ethernet II frame
Destination: FF:FF:FB:3D:28:9C
Source: FF:FF:FE:09:42:A3
Type: 0x0800 (IP)
IPv4 packet
Version: 4
Header length: 20 bytes
DSCP: 0x00
ECN: 0x00
Total length: 43 bytes
IPID: 32181 [could be anything]
Flags: 0x04 (DF bit set, MF bit clear)
Fragment offset: 0
TTL: 64 [could be more or less, depending on the IP stack]
Protocol: 0x11 (UDP)
Source: 172.17.42.37
Destination: 172.17.42.58
UDP packet
Source port: 51500
Destination port: 9
Length: 23 bytes
Data: Hello, world!\r\n
```

c. A computer with IP address `172.17.42.37` initiates a TCP connection from port 51501 to a computer with IP address `172.17.42.58` on port 7. The computer on `.37` sends the string `Hello, world!\r\n` to the peer, which echos back the same message; the two computers then close the connection. Describe each of the IPv4 packets resulting from this conversation.

Because of the way the TCP teardown can work, there are a few different pos-

sibilities for what exactly happens during this conversation; here's one. Note that I haven't used any TCP options here, which keeps the packets simpler; a real conversation between modern hosts is likely to use at least selective acknowledgment and TCP window scaling.

```
IPv4 packet
Version: 4
Header length: 20 bytes
DSCP: 0x00
ECN: 0x00
Total length: 60 bytes
IPID: 51710 [could be anything]
Flags: 0x04 (DF bit set, MF bit clear)
Fragment offset: 0
TTL: 64 [could be more or less, depending on the IP stack]
Protocol: 0x06 (TCP)
Source: 172.17.42.37
Destination: 172.17.42.58
TCP packet
Source port: 51501
Destination port: 7
Sequence number: 1000 [subject to requirements on ISNs]
Acknowledgment number: 0
Header length: 20 bytes
Flags: 0x02 (SYN)
Window size: 5840 [depends on TCP stack and link]
```

```
IPv4 packet
Version: 4
Header length: 20 bytes
DSCP: 0x00
ECN: 0x00
Total length: 60 bytes
IPID: 0 [could be anything]
Flags: 0x04 (DF bit set, MF bit clear)
Fragment offset: 0
TTL: 64 [could be more or less, depending on the IP stack]
Protocol: 0x06 (TCP)
Source: 172.17.42.58
Destination: 172.17.42.37
TCP packet
Source port: 7
Destination port: 51501
```

```
Sequence number: 2000 [subject to requirements on ISNs]
Acknowledgment number: 1001
Header length: 20 bytes
Flags: 0x12 (ACK|SYN)
Window size: 5792 [depends on TCP stack and link]
```

From this point, we only describe source and destination IP addresses, sequence and acknowledgment numbers, TCP flags, window sizes, and data for each packet.

```
Source: 172.17.42.37
Destination: 172.17.42.58
Sequence number: 1001
Acknowledgment number: 2001
Flags: 0x10 (ACK)
Window size: 5856
```

At this point, the three-way handshake is complete, and a TCP connection has been established. Notice the way the window size has grown – no packets have been dropped, so TCP allows more data to be transferred before the next ACK.

```
Source: 172.17.42.37
Destination: 172.17.42.58
Sequence number: 1001
Acknowledgment number: 2001
Flags: 0x18 (ACK|PSH)
Window size: 5856
Data: Hello, world!\r\n
```

Note that the sequence number for the ACK is reused. Sequence numbers are assigned for each byte of data, since it is data transmissions that need to be acknowledged; empty ACK packets need not be acknowledged, so they do not need to take up sequence number space. The PSH flag is set to tell the remote TCP stack to flush its buffers to the application, since this is the logical end of a transmission; if the transmission encompassed multiple packets, PSH would only be set on the last of these.

```
Source: 172.17.42.58
Destination: 172.17.42.37
Sequence number: 2001
Acknowledgment number: 1016
Flags: 0x10 (ACK)
Window size: 5824
```

Notice the ACK number has jumped to 1016; sequence numbers are assigned per byte of data, and our data was 15 bytes long. Once the conversation starts, the empty ACK packet is strictly not necessary, as the packet can be acknowledged by the next data packet, but most TCP stacks emit these.

```
Source: 172.17.42.58
Destination: 172.17.42.37
Sequence number: 2001
Acknowledgment number: 1016
Flags: 0x18 (ACK|PSH)
Window size: 5824
Data: Hello, world!\r\n
```

The server echos data back to the client.

```
Source: 172.17.42.37
Destination: 172.17.42.58
Sequence number: 1016
Acknowledgment number: 2016
Flags: 0x10 (ACK)
Window size: 5856
```

```
Source: 172.17.42.37
Destination: 172.17.42.58
Sequence number: 1016
Acknowledgment number: 2016
Flags: 0x11 (ACK|FIN)
Window size: 5856
```

The client is telling the server that it has no more data to send. Note that the server can continue to send data to the client; this is known as a "half-open" connection.

```
Source: 172.17.42.58
Destination: 172.17.42.37
Sequence number: 2016
Acknowledgment number: 1017
Flags: 0x11 (ACK|FIN)
Window size: 5824
```

FIN packets need to be acknowledged to complete the tear-down of the connection, so the ACK number has increased despite the fact that no further data has been transmitted. The FIN flag is set to indicate that the server also has no more data to send; again, a FIN from one side is not necessarily followed immediately by a FIN from the other.

```
Source: 172.17.42.37
Destination: 172.17.42.58
Sequence number: 1017
Acknowledgment number: 2017
Flags: 0x10 (ACK)
Window size: 5856
```

This last ACK from the client completes the tear-down of the TCP connection in both directions.

Notice the overhead from the TCP connection; what would have taken two packets to say in UDP has taken 10 packets to say in TCP. TCP is therefore frequently undesirable for conversations like this, consisting of short, discrete messages; on the other hand, if transmitting larger streams of information, or if reliability is an issue, the overhead is less of a problem.

2. *Idle scan.* TCP initial sequence numbers aren't the only numbers that are problematic if they are predictable. There's an interesting technique called "idle scan", implemented in recent versions of `nmap`, that relies on a "zombie" host whose IPID numbers are predictable.

   a. How does this scan work? Why does the zombie host have to be idle? Where do the predictable IPID numbers come in?

      The scanning host first sends a packet to the zombie, and looks at the reply to collect its current IPID number. The scanning host sends a TCP SYN packet with source address forged to be that of the zombie host to the scan target. If the port is open on the scan target, the target will reply to the zombie with a SYN|ACK, which will cause the zombie to reply with an RST packet; this causes the IPID number of the zombie's packets to increase in the predictable manner previously detected. If the port is closed, an RST will be sent to the zombie, which silently drops the packet; the zombie's IPID number does not increase in this scenario. Hence, by querying the zombie's IPID number before and after this sequence, we may determine whether or not the queried port was open.

      This scheme breaks down if the zombie has any traffic other than the scan traffic, because increases in the IPID number will then not necessarily be correlated to whether the port on the target is open or not.

   b. From where does the scan appear to be coming from, the scanning host or the zombie? Why? Why might this be a problem if a zombie on your network is being used to scan one of your machines?

      The scan appears to be coming from the zombie, since this is the source address the scan target sees. This means that the scan results reflect the perspective of the zombie! If an attacker can find a zombie on your network, then he can

scan your machines as if he were inside your network, thus possibly evading some of your firewall rules.

c. What can you do to prevent idle scans from being launched from inside your network?

   Since conducting an idle scan depends on the ability to send packets with spoofed source addresses, you can prevent idle scans of hosts outside your network from being launched inside your network by implementing egress filtering of packets with clearly spoofed source addresses (i.e. source addresses outside your network).