Chapter 3: 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 deacti- vating 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.

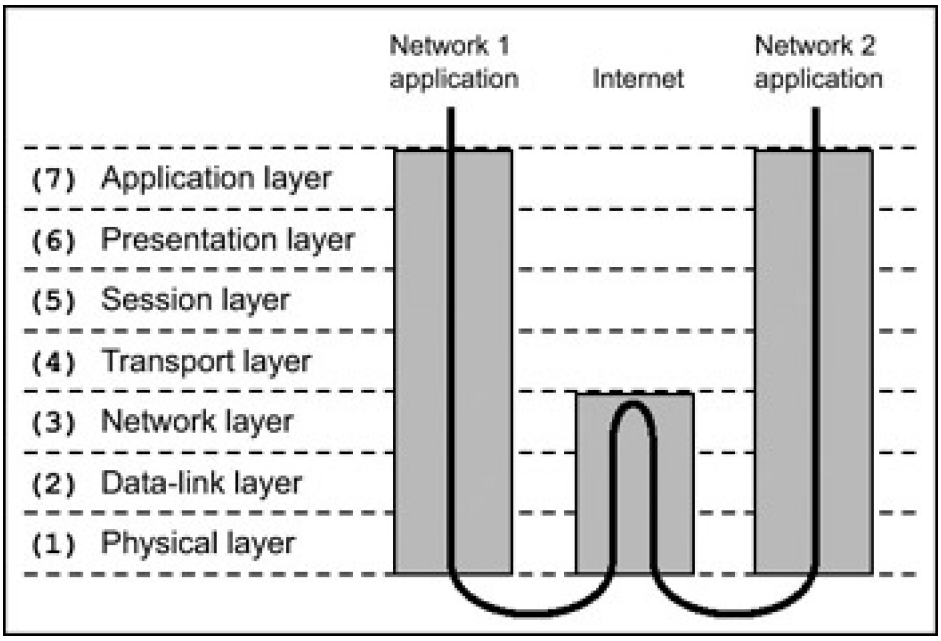
6. **Presentation layer:** This layer is responsible for presenting the data to applications in a syntax or language they understand. This allows for things like encryption and data compression.

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.

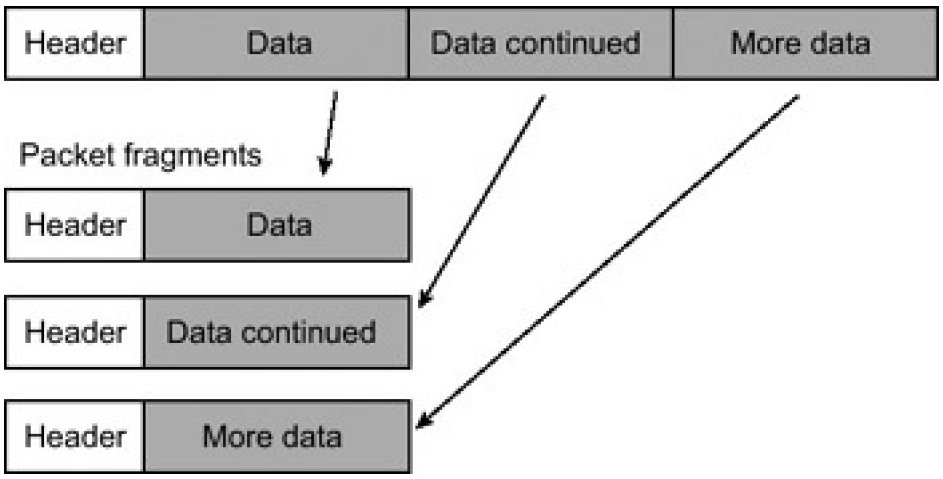
0x321 Network Layer

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

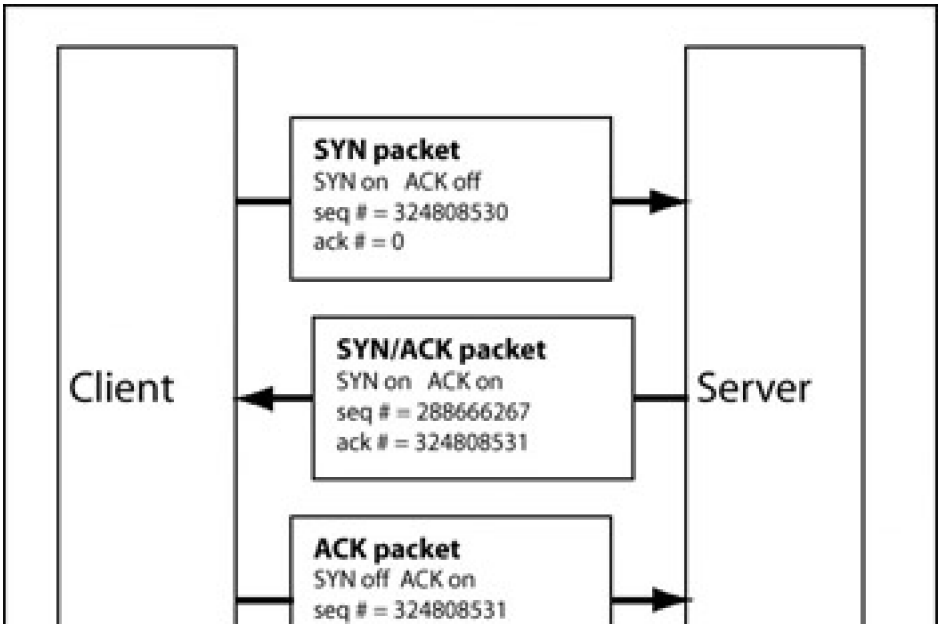
customer wants to return a defective piece of merchandise, they might have to send a message requesting an RMA (Return Material Authorization) number. Then the receptionist would follow the return protocol, ask for a receipt, and eventually issue an RMA number so the customer can mail the product in.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

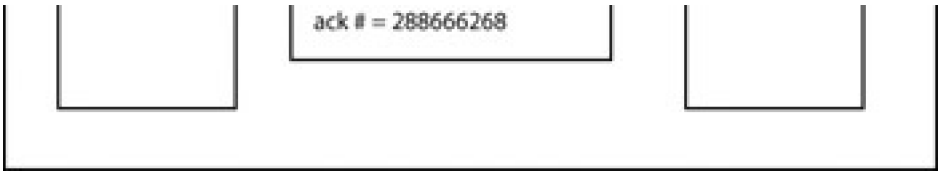
The two major protocols in this layer are Transport Control Protocol (TCP) and User Datagram Protocol (UDP). TCP is the most commonly used protocol for services on the Internet: Telnet, HTTP (web traffic), SMTP (email traffic), and FTP (file transfers) all use TCP. One of the reasons for TCP's popularity 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.

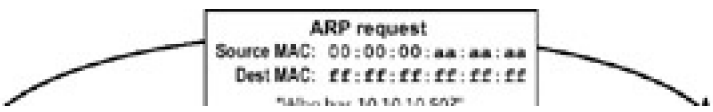
Ethernet exists on this layer, and the layer provides a standard addressing system for all Ethernet devices. These addresses are known as Media Access Control (MAC) addresses. Every Ethernet device is assigned a globally unique address consisting of six bytes, usually written in hexadecimal in the form 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.

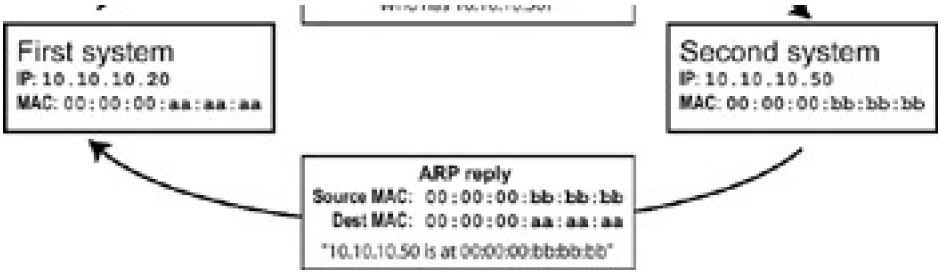
Ethernet headers contain a source address and a destination address, which are used to route Ethernet packets. Ethernet addressing also has a special broadcast address, consisting of all binary 1s (ff:ff:ff:ff:ff:ff). Any Ethernet packet sent to this address will be sent to all the connected devices.

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.





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.

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.

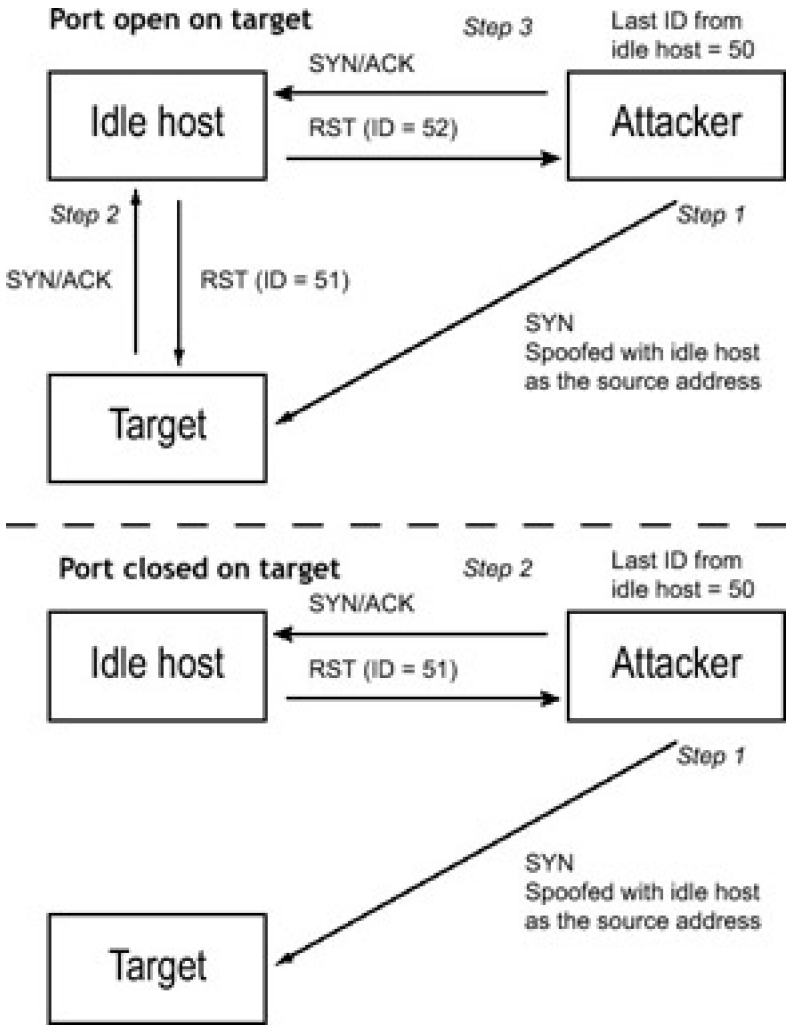
First the attacker gets the current IP ID of the idle host by contacting it with a SYN packet or an unsolicited SYN/ACK packet, and observing the IP ID of the response. By repeating this process a couple more times, the increment that the IP ID changes with each packet can be determined.

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