1

Mitnick Attack

THE BEST WAY TO LEAP into the subject of intrusion detection and hit the ground running is to consider one of the most famous intrusion cases that has ever occurred, when Kevin Mitnick successfully attacked Tsutomu Shimomura's system. This attack will allow us to consider two techniques that are still quite effective today, and we will be able to identify many of the important issues related to intrusion detection for future discussion.

Our source for this information is drawn from Shimomura's post on the subject found at `tsutomu@ariel.sdsc.edu` (Tsutomu Shimomura), `comp.security.misc` Date: 25 Jan 1995.

Exploiting TCP

The techniques Mr. Mitnick used were technical in nature and exploited weaknesses in TCP that were well-known in academic circles, but not considered by system developers. The attack used two techniques: SYN flooding and TCP hijacking. The SYN flood kept one system from being able to transmit. While it was in a mute state, the attacker assumed its apparent identity, and hijacked the TCP connection. Mitnick was able to detect a trust relationship between two computers and exploit that relationship. Nothing has changed since 1994; computer systems are still set up to be over-trusting, often as a convenience to the system administrators.

TCP Review

In order to understand both SYN flooding and TCP hijacking, we need to review some of the characteristics of the *Transport Control Protocol* (TCP) that the attack exploits.

To establish a TCP connection, the two parties execute the three-way handshake as shown below, where A wants to establish a session with B.

1. A sends a SYN packet.
2. B says, "Sure, why not," and acknowledges A's SYN. This is called a SYN/ACK.
3. A says, "Sure I am sure. Let's talk." Then A sends back an acknowledgement of B's SYN/ACK and a connection is established.

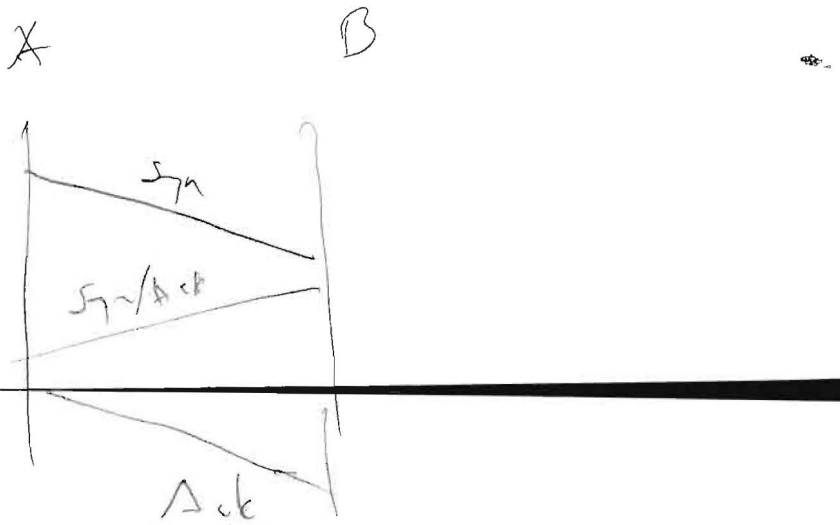
Many of the attacks and probing techniques that attackers use every day are based on intentionally not completing the three-way handshake. The weakness of TCP that Mitnick exploited comes from the early implementations of TCP stacks.

We call the communications software a protocol stack because the various programs that read and write the packets are layered on top of one another. To an application program such as `FTP` or `Telnet`, sockets are the lowest layer, a programming interface to networking hardware. IP is another layer and is above sockets. TCP sits on top of IP. Since TCP is connection oriented, it has to keep state information, including window and sequence number information.

TCP's Roots

When TCP was being developed, you couldn't purchase much memory for machines. If you could get 4MB on a server, you were doing quite well. Therefore, the implementers of IP protocol stacks were very conservative. The following quote was obtained from `http://www.ie.cuhk.edu.hk/~shlam/cstdi/history.html`.

"The Internet is an outgrowth of a project from the 1970s by the U.S. Department of Defense *Advanced Research Projects Agency* (ARPA). The ARPANET, as it was then called, was designed to be a nonreliable network service for computer communications over a wide area. In 1973 and 1974, a standard networking protocol, a communications protocol for exchanging data between computers on a network, emerged from the various research and educational efforts involved in this project. This became known as TCP/IP or the IP suite of protocols. The TCP/IP protocols enabled ARPANET computers to communicate irrespective of their computer operating system or their computer hardware."



In a typical Internet protocol stack, there is information relating to sockets (sockets are a programming interface to networking hardware). There is the IP layer information. TCP is connection oriented or stateful, so the server must keep track of all condition states and sequence numbers.

```
struct ip {
#if defined(bsd)
                u_char  ip_hl:4,          /* header length */
                ip_v:4;                   /* version */
#endif
#if defined(powerpc)
               u_char  ip_v:4,           /* version */
               ip_hl:4;                  /* header length */
#endif
       u_char  ip_tos;                   /* type of service */
       short   ip_len;                   /* total length */
       u_short ip_id;                    /* identification */
       short   ip_off;                   /* fragment offset field */
#define IP_DF 0x3000                     /* dont fragment flag */
#define IP_MF 0x4000                     /* more fragments flag */
       u_char  ip_ttl;                   /* time to live */
       u_char  ip_p;                     /* protocol */
       u_short ip_sum;                   /* checksum */
       struct  in_addr ip_src, ip_dst;   /* source and dest address */
};
```

The header file fragment above is taken from an IP header file on a SunOS 4.1.3 system. A struct, in this case `struct ip`, can be thought of as a database record and the items inside as fields for that record. Every time a new connection is processed, these structs have to be created for socket, ip, and other protocol information. That takes memory, and lots of it. Since memory is finite and was particularly limited during the early days of IP network implementation, limits had to be set. The SYN flood attack exploits the limit of the number of connections that are waiting to be established for a particular service.

SYN Flooding

When an attacker sets up a SYN flood, he has no intention of completing the three-way handshake and establishing the connection. Rather, the goal is to exceed the limits that are set for the number of connections that are waiting to be established for a given service. This can cause the system under attack to be unable to establish any additional connections for that service until the number of waiting connections drops below the threshold. Until the threshold limit is met, each SYN packet generates a SYN/ACK that stays in the queue, which is generally between five and ten total connections, waiting to be established.

There is a timer for each connection, limiting how long the system will wait for the connection to be established. The hourglass in Figure 1.1 represents the timer that tends to be set for about a minute. When the time limit is exceeded, the memory that holds the state for that connection is released, and the service queue count is decremented by one. Once the limit has been reached, the service queue can be kept full, preventing the system from establishing new connections on that port with about 10 new SYN packets per minute.

Covering His Tracks

Since the purpose of the technique is only to write, it doesn't make sense to use the attacker's actual Internet address. The attacker isn't establishing a connection; he is flooding a queue, so there is no point in having the SYN/ACKs return to the attacker. The attacker doesn't want to make it easy for folks to track the connection back to him. Therefore, the source address of the packet is generally spoofed. The IP header below is from actual attack code for a SYN flood. At the bottom, please notice the `dadd` and `sadd` for destination and source address respectively.

```
/* Fill in all the IP header information */
        packet.ip.version=4;            /* 4-bit Version */
        packet.ip.ihl=5;                /* 4-bit Header Length */
        packet.ip.tos=0;                /* 8-bit Type of service */
        packet.ip.tot_len=htons(40);    /* 16-bit Total length */
        packet.ip.id=getpid();          /* 16-bit ID field */
        packet.ip.frag_off=0;           /* 13-bit Fragment offset */
        packet.ip.ttl=255;              /* 8-bit Time To Live */
        packet.ip.protocol=IPPROTO_TCP; /* 8-bit Protocol */
        packet.ip.check=0;             /* 16-bit Header checksum (filled in below)
*/
        packet.ip.saddr=sadd;           /* 32-bit Source Address */
        packet.ip.daddr=dadd;           /* 32-bit Destination Address */
```

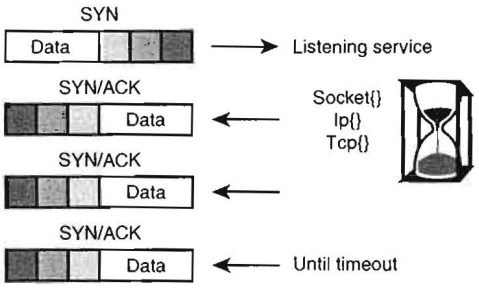


Figure 1.1 Getting down to it.

As you will see in the following code fragment, this technique even uses an error checking routine to make sure the address chosen is routable to, but not active. When the attacker enters an address, the attack code pings the address to ensure it meets these requirements. If the address was active, it would send a RESET when it received the SYN/ACK for the system that is under attack. When the target system received the RESET, it would release the memory and decrement the service queue counter rendering the attack ineffective. From an intrusion detection standpoint, bogus packets that are assembled for the purpose of attacking and probing can be called *crafted packets*. Quite often, the authors of software that craft packets make a small error or take a shortcut, and this gives the packet a unique signature. We can use these signatures in intrusion detection. When we detect evidence of a crafted packet, we know the sender is up to something.

```
case 3:
                                if(!optflags[1]){
                                        fprintf(stderr,"Um, enter a host first\n");
                                        usleep(MENUSLEEP);
                                        break;
                                }
                                                /* Raw ICMP socket */

if((sock2=socket(AF_INET,SOCK_RAW,IPPROTO_ICMP))<0){
                                                 perror("\nHmmm.... socket problems\n");
                                                 exit(1);
                                }
                                printf("[number of ICMP_ECHO's]-> ");
                                fgets(tmp,MENUBUF,stdin);
                                if(!(icmpAmt=atoi(tmp)))break;
                                if(slickPing(icmpAmt,sock2,unreach)){
                                        fprintf(stderr,"Host is reachable... Pick
➥a new one\n");
                                        sleep(1);
```

Now we have a technique that can be used as a generic denial of service; we hit a target system with SYNs until it is unable to speak (establish new connections). Systems that are vulnerable to this attack can be kept out of service until the attacker decides to go away and SYN no more. In the Mitnick attack, the goal was to silence one side of a TCP connection and masquerade as the silenced, trusted party.

Identifying Trust Relationships

So how did Mitnick identify which system to silence? How was he able to determine that there was a trust relationship? It turns out that many complex attacks are preceded by intelligence-gathering techniques, or "recon probes." Here are the recon probes detected by `TCPdump`, a network monitoring tool developed by the Department of Energy's Lawrence Livermore Lab, and reported in Tsutomu's post.

The IP spoofing attack started at about 14:09:32 PST on 12/25/94. The first probes were from `toad.com` (this info derived from packet logs):

```
14:09:32 toad.com# finger -l @target
14:10:21 toad.com# finger -l @server
14:10:50 toad.com# finger -l root@server
14:11:07 toad.com# finger -l @x-terminal
14:11:38 toad.com# showmount -e x-terminal
14:11:49 toad.com# rpcinfo -p x-terminal
14:12:05 toad.com# finger -l root@x-terminal
```

Each of the commands shown, `finger`, `showmount`, and `rpcinfo`, can provide information about UNIX systems. If you work in a UNIX environment and haven't experimented with these commands in a long while, it might be worthwhile to substitute some of your machine names for target, server, and x-terminal to see what can be learned.

- **`finger`** will tell you who is logged on to the system, when they logged on, when they last logged in, where they are logging in from, how long they have been idle, if they have mail, and when their birthday is...well, scratch the birthday. The analogous command for MS Windows systems is NBTSTAT.

```
finger example:
[root@toad /tmp]# finger @some.host.net
[some.host.net]
Login    Name                TTY         Idle    When         Where
chap     Bill Chapman x1568  pts/6       3:11    Tue 17:26    picard
chap     Bill Chapman x1568  console     8:39    Mon 14:44    :0
[root@toad /tmp]#
```

- **`showmount -e`** will provide information about the file systems that are mounted with NFS (Network File System). Of particular interest to attackers are file systems that are mounted world readable or writable—that is, available to everyone.

```
showmount example:
[root@toad /tmp]# showmount -e some.host.net
Export list for some.host.net:
/usr       export-hosts
/usr/local export-hosts
/home      export-hosts
[root@toad /tmp]#
```

- **`rpcinfo`** provides information about the remote procedure call services that are available on a system. `rpcinfo -p` gives the ports where these services reside.

```
rpcinfo  example:
[root@toad /tmp]# rpcinfo -p some.host.net
   program vers proto   port
    100000    3   udp    111  rpcbind
    100000    2   udp    111  rpcbind
    100003    2   udp   2049  nfs
    100024    1   udp    774  status
    100024    1   tcp    776  status
```

```
100021    1   tcp    782  nlockmgr
100021    1   udp    784  nlockmgr
100005    1   tcp   1024  mountd
100005    1   udp   1025  mountd
391004    1   tcp   1025
391004    1   udp   1026
100001    1   udp   1027  rstatd
100001    2   udp   1027  rstatd
100008    1   udp   1028  walld
100002    1   udp   1029  rusersd
100011    1   udp   1030  rquotad
100012    1   udp   1031  sprayd
100026    1   udp   1032  bootparam
```

These days, most sites block TCP port 79 (finger) at their firewall or filtering router, but it might be a good idea to try this from your home ISP account—*get permission first!* Again, hopefully your site blocks TCP/UDP port 111 (portmapper), but this is worth testing as well. In recent years, so-called secure portmappers have become available, either from vendors or as an external package developed by Wietse Venema, available from the Coast archive at FTP://coast.cs.purdue.edu/pub.

Examining Network Traces

In the case of the Mitnick attack, however, none of these ports was blocked, and toad.com was able to acquire information used in the next phase of the attack. From Tsutomu's post:

> We now see 20 connection attempts from apollo.it.luc.edu to x-terminal.shell. The purpose of these attempts is to determine the behavior of x-terminal's TCP sequence number generator. Note that the initial sequence numbers increment by one for each connection, indicating that the SYN packets are *not* being generated by the system's TCP implementation. This results in RSTs conveniently being generated in response to each unexpected SYN/ACK, so the connection queue on x-terminal does not fill up.

As we examine the following TCPdump trace, note how it is in sets of three packets, a SYN from apollo to x-terminal, a SYN/ACK, step two of the three-way handshake, and a RESET from apollo to x-terminal to keep from SYN flooding x-terminal.

How to Read TCPdump Traces:

```
Timestamp      Source host.Source Port   > Dst host.Dst Port: TCP FLAG(s)
14:18:25.906002 apollo.it.luc.edu.1000   > x-terminal.shell: S
SEQ NUM: ACK NUM        TCP Window Size
1382726990:1382726990(0) win 4096
```

Note that in the traces below, +++'s have been added to emphasize the packet triplets.

```
+++
   14:18:25.906002 apollo.it.luc.edu.1000 > x-terminal.shell: S
   1382726990:1382726990(0) win 4096

   14:18:26.094731 x-terminal.shell > apollo.it.luc.edu.1000: S
   2021824000:2021824000(0) ack 1382726991 win 4096

   14:18:26.172394 apollo.it.luc.edu.1000 > x-terminal.shell: R
   1382726991:1382726991(0) win 0
   +++

   +++
   14:18:26.507560 apollo.it.luc.edu.999 > x-terminal.shell: S
   1382726991:1382726991(0) win 4096

   14:18:26.694691 x-terminal.shell > apollo.it.luc.edu.999: S
   2021952000:2021952000(0) ack 1382726992 win 4096

   14:18:26.775037 apollo.it.luc.edu.999 > x-terminal.shell: R
   1382726992:1382726992(0) win 0
   +++
```

In the previous trace, notice the bolded value: This is the sequence number if we take the second set of packets and focus on the sequence number in x-terminal's SYN/ACK; it is 2021952000. The sequence number in the preceding set's SYN/ACK is 2021824000. If we subtract 2021952000 from 2021824000, we get 128,000. Is this of any value? It is if it is repeatable. Let's check one more set of packets.

```
   +++

   14:18:27.014050 apollo.it.luc.edu.998 > x-terminal.shell: S
   ↪1382726992:1382726992(0) win 4096

   14:18:27.174846 x-terminal.shell > apollo.it.luc.edu.998: ack 1382726993 S
   ↪2022080000:2022080000(0) win 4096

   14:18:27.251840 apollo.it.luc.edu.998 > x-terminal.shell: R
   ↪1382726993:1382726993(0) win 0

   14:18:27.544069 apollo.it.luc.edu.997 > x-terminal.shell: S
   ↪1382726993:1382726993(0) win 4096

   14:18:27.714932 x-terminal.shell > apollo.it.luc.edu.997: ack 1382726994 S
   ↪2022208000:2022208000(0) win 4096

   14:18:27.794456 apollo.it.luc.edu.997 > x-terminal.shell: R
   ↪1382726994:1382726994(0) win 0
```

Again, 2022208000–2022080000 = 128,000. So it is repeatable, or perhaps a better word would be predictable. We know that any time we send a SYN to x-terminal, the SYN/ACK will come back 128,000 or higher, as long as it is the next connection. With the ability to silence one side of the TCP connection and trust relationship, and the ability to determine what the sequence number will be, we are almost ready to hijack the connection. Figure 1.2 shows the basic approach.

The Hijack

How can this be possible? Surely the computers would notice that the attacker has the wrong IP address. The IP address is checked for the trust relationship when the connection is being established. Well at least it is checked if the host computer is running software like `TCP wrappers` (also available from the Coast archive). `TCP wrappers` defaults to something called "paranoid mode." As a connection is being established, the computer compares the results of DNS Name Lookup to the results of a DNS Address Lookup system and makes sure the address and name of the connecting system match. If they don't, it drops the connection. The MAC address is not examined because that is the value of the last router, and if the MAC changes during the connection, the host computers will not detect this fact. We have just raised several important points, so let's make sure that we highlight them.

- Checking things only once is a general problem in computer security. One of the primary classes of attacks on a host computer is to allow a program to validate the ownership or permissions of a file (once) and then to quickly introduce a different file before the program notices.
- If the MAC addresses change during a TCP connection, this could be an indicator of TCP hijacking, because the attack may be coming from a new direction and could be detected by intrusion detection systems.
- Computers that are not protected by paranoid mode, doing both forward and reverse DNS lookups when establishing a connection, are fairly vulnerable to being spoofed.

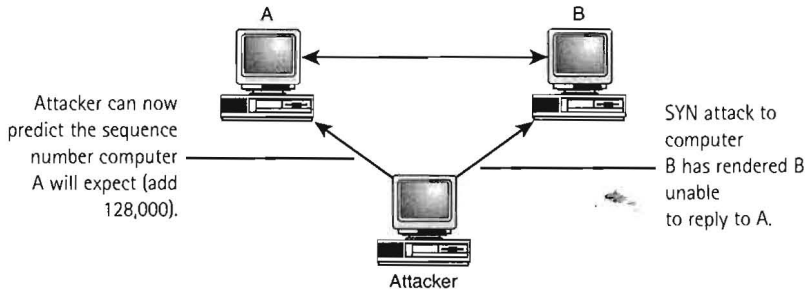


Figure 1.2 Ready for the Kill.

Now we have talked all around the problem, but we still haven't shown why the computers do not realize why the IP address is now being forged or spoofed during the connection. The answer turns out to be quite simple: The Internet address is in the IP header and the sequence number is in the TCP header. TCP applications simply keep track of the sequence number. If you send a packet with the wrong sequence number, the other side will send a RESET and break off the connection. This is why it matters that in the Mitnick attack, x-terminal has a predictable sequence number. So now we can silence one party (server) and make the other party (x-terminal) believe we are that party (server). What happens next? Again, we return to Tsumoto's post:

> We now see a forged SYN (connection request), allegedly from `server.login` to `x-terminal.shell`. The assumption is that x-terminal probably trusts server, so x-terminal will do whatever server (or anything masquerading as a server) asks x-terminal then replies to server with a SYN/ACK, which must be ACK'd in order for the connection to be opened. As server is ignoring packets sent to `server.login`, the ACK must be forged as well.

Normally, the sequence number from the SYN/ACK is required in order to generate a valid ACK. However, the attacker is able to predict the sequence number contained in the SYN/ACK based on the known behavior of x-terminal's TCP sequence number generator and is thus able to ACK the SYN/ACK without seeing it:

```
14:18:36.245045 server.login > x-terminal.shell: S 1382727010:1382727010(0) win
➥4096
14:18:36.755522 server.login > x-terminal.shell: . ack 2024384001 win 4096
```

Here, Mitnick exploits the trust relationship between x-terminal and server. The SYN packet is sent with a spoofed source address. The attacker sends this packet blindly; there is no way for the attacker to see the reply (short of a sniffer planted on x-terminal or server's network). Because Mitnick has used a fake source address, that of server, the SYN/ACK will be sent to server. Because the server knows that it never sent a SYN packet, a request to open a connection, the proper response for server is to send a RESET and break off the connection. However, that isn't going to happen. As shown below, 14 seconds before the main part of the attack, the server's connection queue for the login port is filled with a SYN flood. The server is unable to speak. The proper response is a RESET:

```
14:18:22.516699 130.92.6.97.600 > server.login: S 1382726960:1382726960(0) win
➥4096
14:18:22.566069 130.92.6.97.601 > server.login: S 1382726961:1382726961(0) win
➥4096
14:18:22.744477 130.92.6.97.602 > server.login: S 1382726962:1382726962(0) win
➥4096
14:18:22.830111 130.92.6.97.603 > server.login: S 1382726963:1382726963(0) win
➥4096
```

```
14:18:22.886128 130.92.6.97.604 > server.login: S 1382726964:1382726964(0) win
➥4096
14:18:22.943514 130.92.6.97.605 > server.login: S 1382726965:1382726965(0) win
➥4096
```

The login service is also known as `rlogin` and shell as `rshell`. These are remote "convenience services" that allow access to systems without a pesky password, which can get old if you have to do it often. On UNIX computers, one can generally create a trust relationship for all users except root, or super user, by adding the trusted system and possibly the trusted account in a file called `/etc/hosts.equiv`. A root trusted relationship requires a file called `/.rhosts`. The r-utilities are very obsolete and should not be used anymore; the secure shell service is a far wiser choice since it is harder for the attacker to exploit. In either the `hosts.equiv` or the `.rhost` file, the "+" plus symbol has a special meaning, that of the wildcard. For instance, a `/.rhost` file with a "+ +" means to trust all computers and all users on those computers.

The trusted connection is used to execute the following UNIX command with `rshell —rsh x-terminal "echo + + >>/.rhosts"`. The result of this causes x-terminal to trust as root all computers and all users on these computers as already discussed. That trace is shown below.

```
14:18:37.265404 server.login > x-terminal.shell: P 0:2(2) ack 1 win 4096
14:18:37.775872 server.login > x-terminal.shell: P 2:7(5) ack 1 win 4096
14:18:38.287404 server.login > x-terminal.shell: P 7:32(25) ack 1 win 4096
```

At this point, the connection is terminated by sending a FIN to close the connection. Mr. Mitnick logs in from the computer of his choice and has the ability to execute any command. The target system is compromised.

```
14:18:41.347003 server.login > x-terminal.shell: . ack 2 win 4096
14:18:42.255978 server.login > x-terminal.shell: . ack 3 win 4096
14:18:43.165874 server.login > x-terminal.shell: F 32:32(0) ack 3 win 4096
```

Now, if Mitnick left the computer named server in its mute state and someone tried to `rlogin`, they would fail, which might bring unwanted attention to the situation. So, the connection queue is emptied with a series of RESETs.

We now see RSTs to reset the "half-open" connections and empty the connection queue for `server.login`:

```
14:18:52.298431 130.92.6.97.600 > server.login: R 1382726960:1382726960(0) win
➥4096
14:18:52.363877 130.92.6.97.601 > server.login: R 1382726961:1382726961(0) win
➥4096
14:18:52.416916 130.92.6.97.602 > server.login: R 1382726962:1382726962(0) win
➥4096
14:18:52.476873 130.92.6.97.603 > server.login: R 1382726963:1382726963(0) win
➥4096
14:18:52.536573 130.92.6.97.604 > server.login: R 1382726964:1382726964(0) win
➥4096
```

Right to a Speedy Trial?

At the time of this writing (tonight is December 24, 1998), Kevin Mitnick has been held for approximately 3 years, 10 months, 9 days, 20 hours, 39 minutes, 4 seconds without his day in court.

Detecting the Mitnick Attack

The attack could have been detected by both host-based and network-based intrusion detection systems. It could have been detected at several points from the intelligence-gathering phase all the way to the corruption of `/.rhosts` file, when the target system was fully compromised. Intrusion detection is not a specific tool, but a capability, a blending of tools and techniques. As you move through the material in this book, you will see examples of detects by firewalls and by host-based and network-based intrusion detection systems.

Though the Mitnick attack is several years old, it is still effective. TCP hijacking is still a valuable technique for the more advanced attacker. SYN floods still work on many TCP stacks. Though there are much safer alternatives, such as secure shell, system administrators still use the r-utilities. If we cannot field a capability that allows us to detect the Mitnick attack, what can we detect? To restate, we can use the Mitnick attack as the lowest-level threshold of an intrusion detection capability. A system that cannot reliably detect this attack isn't capable of doing intrusion detection; it is simply something that runs, whirs, and chips, and gives us the warm, numb feeling of security.

Why make such a big deal of this? It turns out that over four years later, TCP hijacking is still almost impossible to reliably detect in the field with a single tool. Various products can demonstrate a detect in a lab, but the number of false alarms (false positives) in the field make this system feature close to useless. The good news is most of the Mitnick attack was trivially detectable, so, let's look at some ways to accomplish this.

Network-Based Intrusion Detection Systems

Network-based intrusion detection systems can reliably detect the entire recon probe trace shown below. As an analyst, you will be tempted to ignore a single finger, but the pattern in entirety really stands out and should never be ignored. Let's consider some of the ways network-based intrusion detection systems might detect this recon probe.

```
14:09:32 toad.com# finger -l @target
14:10:21 toad.com# finger -l @server
14:10:50 toad.com# finger -l root@server
14:11:07 toad.com# finger -l @x-terminal
14:11:38 toad.com# showmount -e x-terminal
14:11:49 toad.com# rpcinfo -p x-terminal
14:12:05 toad.com# finger -l root@x-terminal
```

Early Detects

Early detects are the best detects. Information security researchers and practitioners see the value in taking intelligence-gathering or recon probes very seriously. The Mitnick attack could certainly have been detected at this point.

Trust Relationship

The scan is targeted to exploit a trust relationship. The whole point of the Mitnick probe was to determine the trust relationship between systems. There must have been some earlier intelligence gathering of some form to know what systems to target here. If Mitnick can do this from a network, the site should be able to do the same thing, perhaps even better. Trained analysts who know their networks can often look at an attack to determine if it is a targeted attack, but this doesn't currently exist as a capability of the intrusion detection system.

Port Scan

Intrusion detection systems can usually be configured to watch for a single attacker coming to multiple ports on a host. Port scans are a valuable tool for detecting intelligence gathering. We saw `toad.com` fire three probes to x-terminal. However, two of them, the `showmount` and `rpcinfo`, will probably be directed at the same port, `portmapper`, which is at TCP/UDP 111. Though it is certainly possible to alarm on a port scan of two ports on a single host in less than a minute, real-world traffic will set this alarm off so often that the analyst would certainly increase the threshold to a high enough value that this probe would not be detected as a port scan. It is certainly possible to set the alarm thresholds to report connection attempts to two different ports on a host computer in under a minute. In actual practice, this would create a large number of false alarms. It wouldn't take long for the analyst to give up and set the threshold higher. A network-based system probably would not detect this probe as a port scan.

Host Scan

Host scans happen when multiple systems are accessed by a single system in a short period of time. In our example, `toad.com` connects to three different systems in as many minutes. Host scan detects are an extremely powerful tool that force attackers to coordinate their probes from multiple addresses to avoid detection. In operational experience we have found that one can employ a completely stupid brute force algorithm, flag any host that connects to more than five hosts in an hour, with a very acceptable false positive rate. If you lower the window from an hour to five minutes, connects to three or more hosts will still have a low false positive rate for most sites. If the intrusion detection system can modify the rule for a host scan to eliminate the hosts or conditions that often cause false positives, such as popular Web servers, real audio, and any other broadcast service, then the trip threshold may be able to be set even lower than five per hour and three per five minutes. The host scan detection code in an intrusion detection system should be able to detect the example recon probe.

Connections to Dangerous Ports

The recon probe targets well-known, exploitable ports. For this reason, the recon probe is close to a guaranteed detect. Network-based intrusion detection systems can and do reliably detect connects and attempted connects to SUNRPCs. On the whole, the attacker has some advantages in terms of evading intrusion detection systems; they can go low and slow or they can flood the system with red herring decoys and then go for their actual target. But they probably have to go after a well-known port or service to execute the exploit, and this is where the intrusion detection system has an advantage. SUNRPCs are a very well-known attack point, and every intrusion detection system should be able to detect an attempt against these services.

Host-Based Intrusion Detection Systems

Because the attack was against a UNIX system, we will consider detecting the attack with two types of commonly used UNIX tools, `TCP wrappers` and `tripwire`. `TCP wrappers` logs connection attempts against protected services and can evaluate them against an access control list to determine whether to allow a connection to be made. `tripwire` can monitor the status of individual files and determine whether they were changed. When considering host-based intrusion detection systems, you want at least these capabilities.

TCP wrappers

Of the two tools, `TCP wrappers` is the only one that could reasonably be employed to detect the recon probes. For `TCP wrappers` to work, it would be necessary to edit the `/etc/inetd.conf` file to wrap the services that were probed, such as `finger`. It is also a good idea to add access control lists to `TCP wrappers`. If a system is going to run a service such as `finger`, it is possible to define which systems you will allow to access the `finger` daemon. That way, the access would be logged and the connection would not be permitted. The following is a *fabricated* log entry showing what three `TCP wrappers` finger connection events might look like on a system log facility (`syslog`).

```
Dec  24 14:10:29 target in.finger[11244]: refused connect from toad.com
Dec  24 14:10:35 server in.fingerd[21245]: refused connect from toad.com
Dec  24 14:11:08 x-terminal  in.fingerd[11066]: refused connect from toad.com
```

`IMAP`: A Well-Known Attack Port

On a Web search last week, I found five different `IMAP` (TCP port 143) exploits. Perhaps there are five or even fifty more that haven't been widely released, but the number doesn't matter. `IMAP` lives at port 143, and to port 143 the attacker must go to try his exploit. When the attacker does, he risks being detected by a site with an intrusion detection capability. Though we have confined our discussion to the recon probes, it should be mentioned that the `r-utilities` are also well-known services for hacker attempts. For that reason the attack himself would be easily detected with the capabilities commonly available today.

One of the interesting problems with host-based intrusion detection is how much information to keep and analyze locally and how much to analyze centrally. In the fabricated example, we see that three different systems, target, server, and x-terminal, are reporting to a central log server. A single `finger` attempt logged and evaluated on the host computer might be ignored. However, three finger attempts against three systems might stand out if they were recorded and evaluated on a central or departmental log server.

Access attempts to `portmapper` would be considered higher priority than finger attempts by an analyst. At the time of the Mitnick attack, secure `portmappers` were not widely available. This is no longer the case, and it would be an indication of an archaic or poorly configured UNIX operating system if both logging and access control features were not available for `portmap`. Host-based intrusion detection solutions should certainly detect attempts to access `portmap`.

tripwire

`tripwire` could not reasonably be used to detect the recon probes because it basically creates and stores a high-quality checksum of critical files so that if the file or its attributes change, this fact can be detected. `tripwire` would be able to detect the actual system compromise, the point at which the `/.rhosts` file was overwritten. Unfortunately, even if the alarm goes off in near real-time, it is essentially too late. The system is already compromised, and a scripted attack could rapidly do a lot of damage. This is why early detects are the best detects. If we can detect an intruder in the recon phase of their attack and determine the systems they have an interest in, our chance of detecting the actual attack is improved.

Preventing the Mitnick Attack

Certainly the attack could have been prevented, and at multiple points. A well-configured firewall or filtering router is remarkably inexpensive, easy to configure, and effective at protecting sites from information-gathering probes and attacks originating from the Internet. Even for the timeframe of the attack, this site was left open to more services than was advisable. `http://www.cert.org/FTP/tech_tips/packet_filtering` is a pointer to CERT's recommendations for packet_filtering and although this document was posted after the incident, it does reflect the best practice of that era. Note that finger is not listed as a port to block, but both `SUNRPC` and the `r-utilities` are listed. If the recon probes and `r-utilities` were blocked, it would be much harder for the attacker, perhaps impossible. I would recommend that you download the document and compare it against your site's firewall policy. If you find that your site has a more open filtering policy than that recommended for 1995, your site is probably at risk!

We have already discussed host-based security and the use of access lists. Obviously, systems need to run services in order to accomplish their work efficiently, but it is often possible to specify what systems will be allowed to access a particular service, such as with `TCP wrappers`. In this case, the attacker has to actually compromise a

trusted host and launch the attack from that host. The Mitnick attack simply had to spoof the identity of a trusted host, which is a lot easier than actually compromising the trusted host.

Summary

It is often possible, when doing a post mortem on a successful system compromise or attack, to determine that the attack was preceded by intelligence-gathering "recon" probes. The harder issue is to detect recon probes, take them seriously, and increase the defensive posture of a facility or system. These recon probes will often be used to locate and investigate trust relationships between computer systems.

Attackers will often exploit a trust relationship between two computers. Many times system administrators use such relationships as a convenience for themselves, even though they are aware that this is a "chink in the armor" for the system.

The Mitnick attack deliberately did not complete the TCP three-way handshake in order to SYN flood one side of the trust relationship. Many attacks and probes will intentionally not complete the three-way handshake.

Crafted packets include packets with deliberately false source addresses. These often have a signature that allows intrusion detection to detect their use.

Checking things only once is a general problem in computer security. When designing software or systems, build in the capability to check and then recheck.

The signature of TCP hijacking is that the IP addresses change during a TCP, while the sequence numbers remain correct for the connection. Reliable detection of TCP hijacking is still beyond the reach of single-tool systems in real-world environments.

Intrusion detection is best thought of as a capability, not a single tool. The Mitnick attack can be thought of as an excellent test case. Intrusion detection systems that can not detect this attack on a real-world network with a real-world load (such as a busy T1 or higher) simply mislead their users into thinking they are doing intrusion detection when in fact they are blind. Even the best intrusion detection system will be blind to an attack that it is not programmed to detect. Many intrusion detection analysts prefer to use systems that allow them to craft user-defined filters to detect new or unusual attacks. In the next chapter, we will look at examples of user-defined filters.