Global Information Assurance Certification Paper

Kevin Mitnick, Hacking

No one made Kevin Mitnick into a hacker; that was his own choice and responsibility. Raised in the San Fernando Valley near Los Angeles by his mother, Mitnick has been in and out of trouble with the law since 1981. It was then, as a 17-year-old, that he was placed on probation for stealing computer manuals from a Pacific Bell telephone-switching center in Los Angeles. Those who know Mitnick paint a picture of a man obsessed with the power inherent in controlling the nation's computer and telephone networks. Who is this anti-social computer thievery demon? Who might exorcize this demon? How about a young, Japanese-born physicist, computer-security expert, a wizard like Tsutomu Shimomura?

On Christmas Day, 1994, a hacker launched a sophisticated IP Spoofing attack against Tsutomu Shimomura's computers in San Diego. The attack was launched from toad.com in San Francisco, the Toad Hall computer owned by John Gilmore, a founding employee of Sun Microsystems. By an uncanny coincidence, Shimomura spent the day at Toad Hall with his friend, Julia Menapace. Shimomura's pursuit of the hacker led to computers in Marin County where Shimomura's stolen files were found on The Well, Denver, San Jose and finally to Kevin Mitnick, the fugitive hacker, in Raleigh, North Carolina. Two different attack mechanisms were used. IP source address spoofing and TCP sequence number prediction were used to gain initial access to a diskless workstation being used mostly as an X terminal. After root access had been obtained, an existing connection to another system was hijacked by means of a loadable kernel STREAMS module.
These are some of the machines that figured in Mitnick's hacking. For those who don't know, Telnet is a UNIX process that allows remote log-in to a networked computer, and is one of the main tools a hacker uses on the Net: http is the protocol for serving World Wide Web pages.

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

The apparent purpose of these probes was to determine if there might be some kind of trust relationship amongst these systems that could be exploited with an IP spoofing attack. The source port numbers for the showmount and rpcinfo indicate that the attacker is root on toad.com. About six minutes later, we see a flurry of TCP SYNs (initial connection requests) from 130.92.6.97 to port 513 (login) on server. The purpose of these

SYNs is to fill the connection queue for port 513 on server with "half-open" connections so it will not respond to any new connection requests. In particular, it will not generate TCP RSTs in response to unexpected SYN-ACKs.

As port 513 is also a "privileged" port (< IPPORT_RESERVED), server.login can now be safely used as the putative source for an address spoofing attack on the UNIX "r-services" (rsh, rlogin). 130.92.6.97 appears to be a random (forged) unused address (one that will not generate any response to packets sent to it):

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
14:18:23.002715 130.92.6.97.606 > server.login: S 1382726966:1382726966(0) win 4096
14:18:23.103275 130.92.6.97.607 > server.login: S 1382726967:1382726967(0) win 4096

Server generated SYN-ACKs for the first eight SYN requests before the connection queue filled up. server will periodically retransmit these SYN-ACKs as there is nothing to ACK them. Below, we see 20 connection attempts, only the first nine are shown, 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:

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 382726991:1382726991(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
14:18:26.775395 apollo.it.luc.edu.999 > x-terminal.shell: R 1382726992:1382726992(0) win 0
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: S 2022080000:2022080000(0) ack 1382726993 win 4096

Note that each SYN-ACK packet sent by x-terminal has an initial sequence number which is 128,000 greater than the previous one. Below, we 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 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

The spoofing machine now has a one-way connection to x-terminal.shell, which appears to be from server.login. It can maintain the connection and send data provided that it can properly ACK any data sent by x-terminal. It sends the following:
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
which corresponds to:14:18:37 server# rsh x-terminal "echo + + >>/.rhosts"
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
Total elapsed time since the first spoofed packet: < 16 seconds.
The spoofed connection is shut down. server.login can again accept connections. After root access had been gained via IP address spoofing, a kernel module named "tap-2.01" was compiled and installed on x-terminal:
x-terminal% modstat
Id Type Loadaddr Size B-major C-major Sysnum Mod Name
1 Pdrv ff050000 1000 59. tap/tap-2.01 alpha
x-terminal% ls -l /dev/tap
crwxrwxrwx 1 root 37, 59 Dec 25 14:40 /dev/tap
This appears to be a kernel STREAMS module which can be pushed onto an existing STREAMS stack and used to take control of a tty device. It was used to take control of an

already authenticated login session to the target.

So how easy was it for Kevin Mitnick to use this technique, let's see. IP-spoofing is a complex technical method of attack. It is made up of several components which include trust-relationship exploitation. The Players required are:

A: Target host
B: Trusted host
X: Unreachable host
Z: Attacking host
(1)2: Host 1 masquerading as host 2

The Figures are:

tick	host a	control	host b
1	A	---SYN--->	B

tick: A tick of time. There is no distinction made as to *how* much time passes between ticks, just that time passes. It's generally not a great deal. **Host a**: A machine participating in a TCP-based conversation.
control: This field shows any relevant control bits set in the TCP header and the direction the data is flowing. **Host b**: A machine participating in a TCP-based conversation. In this case, at the first reference point in time host is sending a TCP segment to host b with the SYN bit on. Unless stated, we are generally not concerned with the data portion of the TCP segment.
Trust Relationships: In the Unix world, trust can be given all too easily. Say you have an account on machine A, and on machine B. To facilitate going between the two with a minimum amount of hassle, you want to setup a full-duplex trust relationship between them. In your home directory at A you create a .rhosts file: `echo "B username" > ~/.rhosts` in your home directory at B you create a .rhosts file: `echo "A username" > ~/.rhosts` (Alternately, root can setup similar rules in /etc/hosts.equiv, the difference being that the rules are host wide, rather than just on an individual basis.) Now, you can use any of the r* commands without that annoying hassle of password authentication. These commands will allow address-based authentication, which will grant or deny access based off of the IP address of the service requestor.
Rlogin: Rlogin is a simple client-server based protocol that uses TCP as it's transport. Rlogin allows a user to login remotely from one host to another, and, if the target machine trusts the other, rlogin will allow the convenience of not prompting for a password. It will instead have authenticated the client via the source IP address. So, from our example above, we can use rlogin to remotely login to A from B (or vice-versa) and not be prompted for a password.
Internet Protocol: IP is the connectionless, unreliable network protocol in the TCP/IP suite. It has two 32-bit header fields to hold address information. IP is also the busiest of all the TCP/IP protocols as almost all TCP/IP traffic is encapsulated in IP datagrams. IP's job is to route packets around the network. It provides no mechanism for reliability or accountability, for that, it relies on the upper layers. IP simply sends out datagrams and hopes they make it intact. If they don't, IP can try to send an ICMP error message back to the source, however this packet can get lost as well. (ICMP is Internet

Control Message Protocol and it is used to relay network conditions and different errors to IP and the other layers.) IP has no means to guarantee delivery. Since IP is connectionless, it does not maintain any connection state information. Each IP datagram is sent out without regard to the last one or the next one. This, along with the fact that it is trivial to modify the IP stack to allow an arbitrarily chosen IP address in the source (and destination) fields make IP easily subvert able.

Transmission Control Protocol : TCP is the connection-oriented, reliable transport protocol in the TCP/IP suite. Connection-oriented simply means that the two hosts participating in a discussion must first establish a connection before data may change hands. Reliability is provided in a number of ways but the only two we are concerned with are data sequencing and acknowledgement. TCP assigns sequence numbers to every segment and acknowledges any and all data segments received from the other end. (ACK's consume a sequence number, but are not themselves ACK'd.) This reliability makes TCP harder to fool than IP.

Sequence Numbers, Acknowledgements and other flags: Since TCP is reliable, it must be able to recover from lost, duplicated, or out-of-order data. By assigning a sequence number to every byte transferred, and requiring an acknowledgement from the other end upon receipt, TCP can guarantee reliable delivery. The receiving end uses the sequence numbers to ensure proper ordering of the data and to eliminate duplicate data bytes. TCP sequence numbers can simply be thought of as 32-bit counters. They range from 0 to 4,294,967,295. Every byte of data exchanged across a TCP connection (along with certain flags) is sequenced. The sequence number field in the TCP header will contain the sequence number of the *first* byte of data in the TCP segment. The acknowledgement number field in the TCP header holds the value of next *expected* sequence number, and also acknowledges *all* data up through this ACK number minus one. TCP uses the concept of window advertisement for flow control. It uses a sliding window to tell the other end how much data it can buffer. Since the window size is 16-bits a receiving TCP can advertise up to a maximum of 65535 bytes. Window advertisement can be thought of an advertisement from one TCP to the other of how high acceptable sequence numbers can be. Other TCP header flags of note are RST (reset), PSH (push) and FIN (finish). If a RST is received, the connection is immediately torn down. RSTs are normally sent when one end receives a segment that just doesn't jive with current connection (we will encounter an example below). The PSH flag tells the receiver to pass all the data is has queued to the application, as soon as possible. The FIN flag is the way an application begins a graceful close of a connection (connection termination is a 4-way process). When one end receives a FIN, it ACKs it, and does not expect to receive any more data (sending is still possible, however).

TCP Connection Establishment: In order to exchange data using TCP, hosts must establish a connection. TCP establishes a connection in a 3-step process called the 3-way handshake. If machine A is running an rlogin client and wishes to connect to an rlogin daemon on machine B, the process is as follows:

1 A ---SYN---> B
2 A <---SYN/ACK--- B
3 A ---ACK---> B

At (1) the client is telling the server that it wants a connection. This is the SYN flag's only purpose. The client is telling the server that the sequence number field is valid, and should be checked. The client will set the sequence number field in the TCP header to it's ISN (initial sequence number). The server, upon receiving this segment (2) will respond with it's own ISN (therefore the SYN flag is on) and an ACKnowledgement of the clients first segment (which is the client's ISN+1). The client then ACK's the server's ISN (3). Now, data transfer may take place.

The ISN and Sequence Number: It is important to understand how sequence numbers are initially chosen, and how they change with respect to time. The initial sequence number when a host is bootstrapped is initialized to 1. (TCP actually calls this variable 'tcp_iss' as it is the initial *send* sequence number. The other sequence number variable, 'tcp_irs' is the initial *receive* sequence number and is learned during the 3-way connection establishment. We are not going to worry about the distinction.) This practice is wrong, and is acknowledged as so in a comment the tcp_init() function where it appears. The ISN is incremented by 128,000 every second, which causes the 32-bit ISN counter to wrap every 9.32 hours if no connections occur. However, each time a connect() is issued, the counter is incremented by 64,000. One important reason behind this predictability is to minimize the chance that data from an older stale incarnation (that is, from the same 4-tuple of the local and remote IP-addresses TCP ports) of the current connection could arrive and foul things up. The concept of the 2MSL wait time applies here, but is beyond the scope of this paper. If sequence numbers were chosen at random when a connection arrived, no guarantees could be made that the sequence numbers would be different from a previous incarnation. If some data that was stuck in a routing loop somewhere finally freed itself and wandered into the new incarnation of it's old connection, it could really foul things up.

Ports: To grant simultaneous access to the TCP module, TCP provides a user interface called a port. Ports are used by the kernel to identify network processes, these are strictly transport layer entities (that is to say that IP could care less about them).
Together with an IP address, a TCP port provides an endpoint for network communications. In fact, at any given moment *all* Internet connections can be described by 4 numbers: the source IP address and source port and the destination IP address and destination port. Servers are bound to 'well-known' ports so that they may be located on a standard port on different systems. For example, the rlogin daemon sits on TCP port 513. IP-spoofing consists of several steps, which I will briefly outline here, then explain in detail. First, the target host is chosen. Next, a pattern of trust is discovered, along with a trusted host. The trusted host is then disabled, and the target's TCP sequence numbers are sampled. The trusted host is impersonated, the sequence numbers guessed, and a connection attempt is made to a service that only requires address-based authentication. If successful, the attacker executes a simple command to leave a backdoor.

Required Items: There are a couple of items that are required to wage this attack: target host, trusted host, attacking host (with root access), and IP-spoofing software. Generally the attack is made from the root account on the attacking host against the root account on the target. One often overlooked, but critical factor in IP-spoofing is the fact that the attack is blind. The attacker is going to be taking over the identity of a trusted host in order to subvert the security of the target host. The trusted host is disabled using

the method described below. As far as the target knows, it is carrying on a conversation with a trusted pal. In reality, the attacker is sitting off in some dark corner of the Internet, forging packets from this trusted host while it is locked up in a denial of service battle. The IP datagrams sent with the forged IP-address reach the target fine (recall that IP is a connectionless-oriented protocol-- each datagram is sent without regard for the other end) but the datagrams the target sends back (destined for the trusted host) end up in the bit-bucket. The attacker never sees them. The intervening routers know where the datagrams are supposed to go. They are supposed to go the trusted host. As far as the network layer is concerned, this is where they originally came from, and this is where responses should go. Of course once the datagrams are routed there, and the information is demultiplexed up the protocol stack, and reaches TCP, it is discarded (the trusted host's TCP cannot respond. So the attacker has to be smart and *know* what was sent, and *know* what response the server is looking for. The attacker cannot see what the target host sends, but she can *predict* what it will send; that coupled with the knowledge of what it *will* send, allows the attacker to work around this blindness. After a target is chosen the attacker must determine the patterns of trust (for the sake of argument, we are going to assume the target host *does* in fact trust somebody. If it didn't, the attack would end here). Figuring out who a host trusts may or may not be easy. A 'showmount -e' may show where file systems are exported, and rpcinfo can give out valuable information as well. If enough background information is known about the host, it should not be too difficult. If all else fails, trying neighboring IP addresses in a brute force effort may be a viable option. Once the trusted host is found, it must be disabled. Since the attacker is going to impersonate it, she must make sure this host cannot receive any network traffic and foul things up. There are many ways of doing this, the one I am going to discuss is TCP SYN flooding. A TCP connection is initiated with a client issuing a request to a server with the SYN flag on in the TCP header. Normally the server will issue a SYN/ACK back to the client identified by the 32-bit source address in the IP header. The client will then send an ACK to the server (as we saw in figure 1 above) and data transfer can commence. There is an upper limit of how many concurrent SYN requests TCP can process for a given socket, however. This limit is called the backlog, and it is the length of the queue where incoming (as yet incomplete) connections are kept. This queue limit applies to both the number of incomplete connections (the 3-way handshake is not complete) and the number of completed connections that have not been pulled from the queue by the application by way of the accept() system call. If this backlog limit is reached, TCP will silently discard all incoming SYN requests until the pending connections can be dealt with. Therein lies the attack. The attacking host sends several SYN requests to the TCP port it desires disabled. The attacking host also must make sure that the source IP-address is spoofed to be that of another, currently unreachable host (the target TCP will be sending it's response to this address. (IP may inform TCP that the host is unreachable, but TCP considers these errors to be transient and leaves the resolution of them up to IP (reroute the packets, etc) effectively ignoring them.) The IP-address must be unreachable because the attacker does not want any host to receive the SYN/ACKs that will be coming from the target TCP (this would result in a RST being sent to the target TCP, which would foil our attack). The process is as follows:

```
1     Z(x)   ---SYN--->     B
      Z(x)   ---SYN--->     B
      Z(x)   ---SYN--->     B
      Z(x)   ---SYN--->     B
      Z(x)   ---SYN--->     B
2     X   <---SYN/ACK---     B
      X   <---SYN/ACK---     B
3     X     <---RST---      B
```

At (1) the attacking host sends a multitude of SYN requests to the target (remember the target in this phase of the attack is the trusted host) to fill it's backlog queue with pending connections. (2) The target responds with SYN/ACKs to what it believes is the source of the incoming SYNs. During this time all further requests to this TCP port will be ignored. Different TCP implementations have different backlog sizes. BSD generally has a backlog of 5 (Linux has a backlog of 6). There is also a 'grace' margin of 3/2. That is, TCP will allow up to backlog*3/2+1 connections. This will allow a socket one connection even if it calls listen with a backlog of 0. Now the attacker needs to get an idea of where in the 32-bit sequence number space the target's TCP is. The attacker connects to a TCP port on the target (SMTP is a good choice) just prior to launching the attack and completes the three-way handshake. The process is exactly the same as fig(1), except that the attacker will save the value of the ISN sent by the target host. Often times, this process is repeated several times and the final ISN sent is stored. The attacker needs to get an idea of what the RTT (round-trip time) from the target to the host. The process can be repeated several times, and an average of the RTT's is calculated.) The RTT is necessary in being able to accurately predict the next ISN. The attacker has the baseline (the last ISN sent) and knows how the sequence numbers are incremented (128,000/second and 64,000 per connect) and now has a good idea of how long it will take an IP datagram to travel across the Internet to reach the target (approximately half the RTT, as most times the routes are symmetrical). After the attacker has this information, he immediately proceeds to the next phase of the attack (if another TCP connection were to arrive on any port of the target before the attacker was able to continue the attack, the ISN predicted by the attacker would be off by 64,000 of what was predicted). When the spoofed segment makes its way to the target, several different things may happen depending on the accuracy of the attacker's prediction:
If the sequence number is exactly where the receiving TCP expects it to be, the incoming data will be placed on the next available position in the receive buffer. If the sequence number is LESS than the expected value the data byte is considered a retransmission, and is discarded. If the sequence number is GREATER than the expected value but still within the bounds of the receive window, the data byte is considered to be a future byte, and is held by TCP, pending the arrival of the other missing bytes. If a segment arrives with a sequence number GREATER than the expected value and NOT within the bounds of the receive window the segment is dropped, and TCP will send a segment back with the *expected* sequence number.

```
1     Z(b)   ---SYN--->     A
2     B    <---SYN/ACK---    A
```

3 Z(b) ---ACK---> A
4 Z(b) ---PSH---> A

The attacking host spoofs her IP address to be that of the trusted host (which should still be in the death-throes of the D.O.S. attack) and sends it's connection request to port 513 on the target (1). At (2), the target responds to the spoofed connection request with a SYN/ACK, which will make it's way to the trusted host (which, if it *could* process the incoming TCP segment, it would consider it an error, and immediately send a RST to the target). If everything goes according to plan, the SYN/ACK will be dropped by the gagged trusted host. After (1), the attacker must back off for a bit to give the target ample time to send the SYN/ACK (the attacker cannot see this segment). Then, at (3) the attacker sends an ACK to the target with the predicted sequence number (plus one, because we're ACKing it). If the attacker is correct in her prediction, the target will accept the ACK. The target is compromised and data transfer can commence (4). Generally, after compromise, the attacker will insert a backdoor into the system that will allow a simpler way of intrusion. (Often a `cat + + >> ~/.rhosts` is done. This is a good idea for several reasons: it is quick, allows for simple re-entry, and is not interactive. Remember the attacker cannot see any traffic coming from the target, so any responses are sent off into oblivion). IP-Spoofing works because trusted services only rely on network address based authentication. Since IP is easily duped, address forgery is not difficult. The hardest part of the attack is in the sequence number prediction, because that is where the guesswork comes into play. Even a machine that wraps all its incoming TCP bound connections with TCP wrappers, is still vulnerable to the attack. TCP wrappers rely on a hostname or an IP address for authentication. Reduce unknowns and guesswork to a minimum, and the attack has a better chance of succeeding.

CONCLUSION

Although this attack method seems faultless, Kevin Mitnick, the well-known hacker, was arrested on February 15, 1995. His crime spree included the theft of thousands of data files and at least 20,000 credit card numbers from computer systems around the nation. He was caught by his own obsession and proved that crime and punishment online is a lot harder to score than Dungeons-and-Dragons.

REFERENCES

Phrack Magazine, www.fc.net/phrack/files/p48/p48-14.html

Mitnick's Malice, Shimomura's Chivalry, www.salon.com

Evidence, www.takedown.com

The Kevin Mitnick/Tsutomu Shimomura affair, www.gulker.com/ra/hack

The January 1995 Systems Intrusion, www.well.com/intrusion