Stealth Communications – Covert Channels

"O divine art of subtlety and secrecy! Through you we learn to be invisible, through you inaudible; and hence we can hold the enemy's fate in our hands." – Sun Tzu. "The Art of War"

- After attackers have compromised a system, they will install a backdoor into the system to maintain access and to communicate with the system
- In order to avoid detection, an attacker will utilize stealth mechanisms to communicate with the backdoor program on the compromised system.
- These stealth mechanisms are known as "covert channels".
- There are many other uses for covert channels:
    - Ex-employees who had system administrator privileges before being terminated, will install a covert channel to maintain unauthorized access.
    - Spies or employees sending out classified or restricted information to outside organizations.
- A Covert channel is a simple yet very effective mechanism for sending and receiving information data between machines without alerting any firewalls and IDS's on the network.
- The technique derives its stealthy nature by virtue of the fact that it sends traffic through ports that most firewalls will permit through.
- In addition the technique can bypass an IDS by appearing to be an innocuous packet carrying ordinary information when in fact it is concealing its actual data in one of the several control fields in the TCP and IP headers.
- It is very difficult to protect against these mechanisms because the technique avoids detection by a stateless IDS by using a variety-randomized signatures.
- However, the activity can still be detected by diligently examining network traffic for certain patterns in the protocol information that will characterize the tool being used.
- There are a number of tools available that can be used to implement covert channels, the two most popular and common ones are:
    - **Covert_TCP** : developed and released by Craig Rowland (www.psionic.com)
    - **Loki**: developed and released by "daemon9" (www.phrack.com)

- We will study and analyze **covert_tcp** in detail. This tool provides three different methods of sending covert data embedded within one of the following fields:

    - The IP packet identification field.
    - The TCP initial sequence number field.
    - The TCP acknowledge sequence number field “Bounce”.

- We will examine the first and third methods. The test bed will consist of two machines. One is a passive server (receiver) and the other is a client (transmitter) that initiates a transfer with the server.

- The server would normally be a compromised machine and have the code running on it, listening for connections on any specified port.

- It should be noted that the server need not always be a compromised machine; a legitimate owner of the machine could use this tool to transfer unauthorized material in and out of a network.

- The client will be the machine that initiates a connection with the server on the specified port and sends information to it. Port 80 was used for this experiment though any port may be used.

- Most firewalls will permit traffic through this port since most networks have web servers running on them.

- In addition an IDS (**Snort)** with all the current rule sets will be installed and kept running during the experiments.

- To obtain some correlation we will also use **tcpdump** to capture all packets entering and leaving the server machine.

Embedding within the IP Identification field

- The 16-bit Identification field of the IP header is used to identify fragments that make up a complete packet.

- This method simply encodes the **Identification field** with the ASCII representation of the character to be sent. The packet that carries this information is a connection request (SYN).

- The server end reads the Identification field and converts character to its printable form by dividing the numerical value in the field by 256.

- With the server running, the client machine connected to it on port 80. A string of characters ("**Covert**") was sent from the client to the server.

- None of the rule sets on Snort reported any alerts due to this traffic. All the relevant packets captured by **tcpdump** are shown below.

- The first packet two packets below illustrate the initial transmission from the client to the server and the response from the server machine.

- It can be seen that the Identification field in the first packet translates to **43H**, which is the character "**C**".

- The server program correctly reads the character and stores it, however the TCP/IP stack responds to the SYN packet with an ACK to the previous packet and a reset (RST).

- Since this is a TCP application one would expect the classic three-way handshake following the initial SYN packet. This however is a characteristic of the way the code is implemented and is discussed later.

15:46:44.594323 eth0 < x.x.x.x.39946 > 192.168.1.60.http: S 1966080:1966080(0) win 512 (ttl 55, id 17152)

4500 0028 4300 0000 3706 648f xxxx xxxx
c0a8 013c 9c0a 0050 001e 0000 0000 0000
5002 0200 3529 0000 0000 0000 0000

15:46:44.594323 eth0 > 192.168.1.60.http > x.x.x.x.39946: R 0:0(0) ack 1966081 win 0 (DF) (ttl 255, id 0)

4500 0028 0000 4000 ff06 9f8e c0a8 013c
xxxx xxxx 0050 9c0a 0000 0000 001e 0001
5014 0000 3716 0000

- The next two packets below illustrate the sequence that conveys the next character in the string from the client to the server and the response from the server machine. It can be seen that the Identification field in the first packet translates to **6FH**, which is the character "**o**".

- Just as before the server responds by acknowledging the previous packet and a reset at the same time.

15:46:45.604323 eth0 < x.x.x.x.54296 > 192.168.1.60.http: S 504102912:504102912(0) win 512 (ttl 55, id 28416)

4500 0028 6f00 0000 3706 388f xxxx xxxx
c0a8 013c d418 0050 1e0c 0000 0000 0000
5002 0200 df2c 0000 0000 0000 0000

15:46:45.604323 eth0 > 192.168.1.60.http > x.x.x.x.54296: R 0:0(0) ack 504102913 win 0 (DF) (ttl 255, id 0)

4500 0028 0000 4000 ff06 9f8e c0a8 013c
xxxx xxxx 0050 d418 0000 0000 1e0c 0001
5014 0000 e119 0000

- The rest of the traffic that makes up the string "**Covert**" is shown below:

15:46:46.614323 eth0 < x.x.x.x.14879 > 192.168.1.60.http: S 117964800:117964800(0) win 512 (ttl 55, id 30208)

4500 0028 7600 0000 3706 318f xxxx xxxx
c0a8 013c 3a1f 0050 0708 0000 0000 0000
5002 0200 902a 0000 0000 0000 0000

15:46:46.614323 eth0 > 192.168.1.60.http > x.x.x.x.14879: R 0:0(0) ack 117964801 win 0 (DF) (ttl 255, id 0)

4500 0028 0000 4000 ff06 9f8e c0a8 013c
xxxx xxxx 0050 3a1f 0000 0000 0708 0001
5014 0000 9217 0000

15:46:47.624323 eth0 < x.x.x.x.31780 > 192.168.1.60.http: S 3741384704:3741384704(0) win 512 (ttl 55, id 25856)

4500 0028 6500 0000 3706 428f xxxx xxxx
c0a8 013c 7c24 0050 df01 0000 0000 0000
5002 0200 762b 0000 0000 0000 0000

15:46:47.624323 eth0 > 192.168.1.60.http > x.x.x.x.31780: R 0:0(0) ack 3741384705 win 0 (DF) (ttl 255, id 0)

4500 0028 0000 4000 ff06 9f8e c0a8 013c
xxxx xxxx 0050 7c24 0000 0000 df01 0001
5014 0000 7818 0000

15:46:48.634323 eth0 < x.x.x.x.17925 > 192.168.1.60.http: S 3238723584:3238723584(0) win 512 (ttl 55, id 29184)

4500 0028 7200 0000 3706 358f xxxx xxxx
c0a8 013c 4605 0050 c10b 0000 0000 0000
5002 0200 ca40 0000 0000 0000 0000

15:46:48.634323 eth0 > 192.168.1.60.http > x.x.x.x.17925: R 0:0(0) ack 3238723585 win 0 (DF) (ttl 255, id 0)

4500 0028 0000 4000 ff06 9f8e c0a8 013c
xxxx xxxx 0050 4605 0000 0000 c10b 0001
5014 0000 cc2d 0000

15:46:49.644323 eth0 < x.x.x.x.51999 > 192.168.1.60.http: S 2569076736:2569076736(0) win 512 (ttl 55, id 29696)

4500 0028 7400 0000 3706 338f xxxx xxxx
c0a8 013c cb1f 0050 9921 0000 0000 0000
5002 0200 6d10 0000 0000 0000 0000

15:46:49.644323 eth0 > 192.168.1.60.http > x.x.x.x.51999: R 0:0(0) ack 2569076737 win 0 (DF) (ttl 255, id 0)

4500 0028 0000 4000 ff06 9f8e c0a8 013c
xxxx xxxx 0050 cb1f 0000 0000 9921 0001
5014 0000 6efd 0000

- Examining the above sequence, the most obvious observation is that it is a sequence quite uncharacteristic of a typical TCP sequence.
- Normally one would expect the server side to respond with a SYN and an ACK. Instead, every SYN is responded to with a RST.
- The reason for this becomes apparent if we examine the code. In a normal TCP server application, the program issues ***listen()*** and ***accept()*** calls; the client then issues a ***connect()*** call to initiate a three-way handshake.
- This particular application uses raw sockets, which are usually used for applications that use protocols such as ICMP. Both the server and client functions use blocking ***read()*** calls.

- Thus, the stack receives a datagram for an unbound socket and simply issues a RST. This sequence observed as part of network traffic can therefore be associated with this particular tool.

- It must be noted however that a hacker with a reasonable knowledge of programming will be able to modify the basic experimental tool to make it appear to be a normal and legitimate network application.

- Even though there is no definite signature generated by this application for use in an IDS, there are certain suspicious patterns in the above sequence that would alert an observant analyst to the presence of covert channel.

- The IP Identification field usually increments by one each time the stack transmits a datagram. In the above sequence we see that identification sequence is quite erratic not to mention the fact that is decrementing relative to the timestamps in certain cases.

- Another observation that we can make is that the port numbers from the client are changing with every connection request and within a very short interval. This is very unusual especially considering the fact that the requests are destined for the same destination port.

- Even though this exploit eluded the IDS there are enough telltale symptoms in the network traffic to alert an analyst to presence of a covert channel.

- However to detect this will require the capturing of all network traffic, which will become an onerous task. A better approach would be to identify traffic generated to and from specific IP addresses and just trap that traffic.

Embedding in the TCP Acknowledge Sequence Number Field

- This method spoofs the IP address of the client and bounces the information bearing datagram off a bounce server (i.e., the spoofed IP) thus creating a difficult to detect, anonymous one-way communication channel.

- The basis for this method is the characteristic of the TCP/IP three-way handshake that specifies that a server respond to a connect request (SYN) with a SYN/ACK packet.

- The ACK field contains the original sequence number plus one, i.e., indicating the next sequence number it is expecting.

- In this method the character to be sent to the server is embedded in this field. The initial connection request datagram from the client is crafted such that the destination IP is the spoofed bounce server IP and the source IP is the address of the machine running the passive server code.

- In this way the bounce server receives the initial SYN and responds to the server machine with a SYN/ACK of its own or a RST depending on the status of the port specified in the connection request.

- It essentially relays (albeit unwittingly) the request from the client to the server. It is always a good idea to use a port such as port 80 (http) which will usually have a lot of traffic associated with it, thus providing an excellent means for concealing the covert traffic.

- The listening server will receive the incoming datagram and decode the ACK sequence number back to the original ASCII representation of the character. The sequence number is converted to ASCII by dividing the numerical value of that field by 16777216 (representation of 2^{24}).

- We will use three machines on the same subnet to demonstrate and analyze this exploit. The client machine has IP address 192.168.1.111, the bounce server has IP address 192.168.1.20, and the server machine has IP address 192.168.1.60.

- The first packet two packets below illustrate the initial transmission from the client to the server and the response from the server machine.

- It can be seen that the **Identification field** in the first packet translates to **43H**, which is the character "**C**". The server program correctly reads the character and stores it, however the TCP/IP stack responds to the SYN packet with an ACK to the previous packet and a reset (RST).

- Just as before the active IDS was **Snort** together with **tcpdump** to capture all packets entering and leaving the server machine.

- The client machine was used to send the string “Covert”. None of the rule sets on Snort reported any alerts due to this traffic. All the relevant packets captured by tcpdump are shown below.

- The first packet two packets below illustrate the initial transmission from the bounce server to the covert TCP server and the response from the server machine. It can be seen that the ACK field in the first packet translates to 43H (1124073473/16777216 = 67 = 43h), which is the ASCII character “C”.

19:50:12.957787 eth0 < 192.168.1.20.http > 192.168.1.60.http: S 81531:81531(0) ack 1124073473 win 8576 <mss 1460> (DF) (ttl 128, id 23554)
4500 002c 5c02 4000 8006 1b29 c0a8 0114
c0a8 013c 0050 0050 0001 3e7b 4300 0001
6012 2180 70d8 0000 0204 05b4 0000

19:50:12.957787 eth0 > 192.168.1.60.http > 192.168.1.20.http: R 1124073473:1124073473(0) win 0 (DF) (ttl 255, id 0)
4500 0028 0000 4000 ff06 f82e c0a8 013c
c0a8 0114 0050 0050 4300 0001 0000 0000
5004 0000 e89e 0000

- The server program correctly reads the character and stores it, however the TCP/IP stack responds to the SYN packet with a reset (RST) for reasons similar to those described in the previous method.

- In this case it is appropriate for the TCP/IP stack to respond to an unsolicited SYN/ACK with a RESET. The rest of the traffic for the complete string is shown below.

19:50:13.967787 eth0 < 192.168.1.20.http > 192.168.1.60.http: S 81544:81544(0) ack 1862270977 win 8576 <mss 1460> (DF) (ttl 128, id 23810)
4500 002c 5d02 4000 8006 1a29 c0a8 0114
c0a8 013c 0050 0050 0001 3e88 6f00 0001
6012 2180 44cb 0000 0204 05b4 0000

19:50:13.967787 eth0 > 192.168.1.60.http > 192.168.1.20.http: R 1862270977:1862270977(0) win 0 (DF) (ttl 255, id 0)
4500 0028 0000 4000 ff06 f82e c0a8 013c
c0a8 0114 0050 0050 6f00 0001 0000 0000
5004 0000 bc9e 0000

19:50:14.977787 eth0 < 192.168.1.20.http > 192.168.1.60.http: S 81551:81551(0) ack 1979711489 win 8576 <mss 1460> (DF) (ttl 128, id 24066)

4500 002c 5e02 4000 8006 1929 c0a8 0114
c0a8 013c 0050 0050 0001 3e8f 7600 0001
6012 2180 3dc4 0000 0204 05b4 0000

19:50:14.977787 eth0 > 192.168.1.60.http > 192.168.1.20.http: R 1979711489:1979711489(0) win 0 (DF) (ttl 255, id 0)

4500 0028 0000 4000 ff06 f82e c0a8 013c
c0a8 0114 0050 0050 7600 0001 0000 0000
5004 0000 b59e 0000

19:50:15.987787 eth0 < 192.168.1.20.http > 192.168.1.60.http: S 81552:81552(0) ack 1694498817 win 8576 <mss 1460> (DF) (ttl 128, id 24322)

4500 002c 5f02 4000 8006 1829 c0a8 0114
c0a8 013c 0050 0050 0001 3e90 6500 0001
6012 2180 4ec3 0000 0204 05b4 0000

19:50:15.987787 eth0 > 192.168.1.60.http > 192.168.1.20.http: R 1694498817:1694498817(0) win 0 (DF) (ttl 255, id 0)

4500 0028 0000 4000 ff06 f82e c0a8 013c
c0a8 0114 0050 0050 6500 0001 0000 0000
5004 0000 c69e 0000

19:50:16.997787 eth0 < 192.168.1.20.http > 192.168.1.60.http: S 81563:81563(0) ack 1912602625 win 8576 <mss 1460> (DF) (ttl 128, id 24578)

4500 002c 6002 4000 8006 1729 c0a8 0114
c0a8 013c 0050 0050 0001 3e9b 7200 0001
6012 2180 41b8 0000 0204 05b4 0000

19:50:16.997787 eth0 > 192.168.1.60.http > 192.168.1.20.http: R 1912602625:1912602625(0) win 0 (DF) (ttl 255, id 0)

4500 0028 0000 4000 ff06 f82e c0a8 013c
c0a8 0114 0050 0050 7200 0001 0000 0000
5004 0000 b99e 0000

19:50:18.007787 eth0 < 192.168.1.20.http > 192.168.1.60.http: S 81568:81568(0) ack 1946157057 win 8576 <mss 1460> (DF) (ttl 128, id 24834)

4500 002c 6102 4000 8006 1629 c0a8 0114
c0a8 013c 0050 0050 0001 3ea0 7400 0001
6012 2180 3fb3 0000 0204 05b4 0000

19:50:18.007787 eth0 > 192.168.1.60.http > 192.168.1.20.http: R 1946157057:1946157057(0) win 0 (DF) (ttl 255, id 0)
4500 0028 0000 4000 ff06 f82e c0a8 013c
c0a8 0114 0050 0050 7400 0001 0000 0000
5004 0000 b79e 0000

- Just as before the IDS did not raise any alerts as a result of this traffic so detection of this activity relies on tcpdump captures.

- Examining the above sequence, the most obvious observation is the fact that there is a series of SYN/ACK packets with no evidence of SYN packets (connection requests) that would have elicited such responses.

- This is the key to identifying this method within network traffic. Note that the IP address of the client (192.168.1.111) remains concealed to the server with this method.

- Also note that the source and destination ports are the same. By default both the client and server programs use port 80 as the source port. This setting can be changed to any value, but the fact remains that the source and destination ports will be the same.

- Two high ports sending data to each other are equally as suspicious as two applications on port 80 communicating if not more so. We could of course randomize the client and server ports however we have to use an open port on the bounce server and the most common open port on all servers is port 80.

- Looking at the sequence there are characteristics other than similar port numbers that should cause an analyst to examine the traffic from these addresses very closely.

- For example, within a span of six seconds there have been six SYN/ACK's received and within that set of datagrams the ACK sequence has decremented. This can happen with out of sequence packets but not likely within such a short period of time.

Analysis of traffic at the "Bounce" server

- A separate bouncer server with tcpdump running on it was setup for this experiment.
- The objective was to collect data using tcpdump at the bounce server and attempt to identify network activity generated by a covert channel, and to further illustrate the effectiveness of this exploit in concealing the address of the covert client.
- The address of this bounce server is 192.168.1.10, the covert server is still 192.168.1.60 and the covert client is 192.168.1.111.
- The following sequence of packets illustrates the sequence generated by sending the first character ("C") in the string.

10:11:47.145612 eth0 < 192.168.1.60.http > 192.168.1.10.http: S 1124073472:1124073472(0) win 512 (ttl 64, id 29184)
4500 0028 7200 0000 4006 8539 c0a8 013c
c0a8 010a 0050 0050 4300 0000 0000 0000
5002 0200 e6ab 0000 0000 0000 0000

10:11:47.155612 eth0 > 192.168.1.10.http > 192.168.1.60.http: S 3809931549:3809931549(0) ack 1124073473 win 5840 <mss 1460> (DF) (ttl 64, id 0)
4500 002c 0000 4000 4006 b735 c0a8 010a
c0a8 013c 0050 0050 e316 f11d 4300 0001
6012 16d0 e5d9 0000 0204 05b4

10:11:47.155612 eth0 < 192.168.1.60.http > 192.168.1.10.http: R 1124073473:1124073473(0) win 0 (DF) (ttl 255, id 0)
4500 0028 0000 4000 ff06 f838 c0a8 013c
c0a8 010a 0050 0050 4300 0001 0000 0000
5004 0000 e8a8 0000 0000 0000 0000

- The first packet is the initial SYN from the client (192.168.1.111) to the bounce server but with a spoofed source IP of the covert server (192.168.1.60). Note that the sequence number field has been encoded with the first character we wish to send, translated as described in the previous section.
- The bounce server responds with a SYN/ACK of its own to the covert server thinking that it is the remote end that wishes to establish a connection. The ACK field of this packet contains the encoded character plus one.

- The covert server gets an unsolicited SYN/ACK packet and correctly responds with a reset (RST). The covert server that was listening on port 80 has received the packet and stores the character after translating it back to its ASCII representation.

- The rest of the characters in the string will generate a similar sequence of packets.

- As can be seen in this sequence the address of the covert client does not appear anywhere.

- The one characteristic in the sequence that should alert an analyst to anomalous activity is the fact that a well-known service such as http (port 80) is initiating a connection with port 80 on the bounce server.

- This in itself should be cause for concern and further action. Secondly, the pattern will establish itself quite clearly over time; a connection from the spoofed IP will initiate a connection request, the bounce server will naturally respond with a SYN/ACK but the covert server will always respond with a reset.

- This pattern can be associated with a high degree of certainty to a covert channel and the use of the machine as a bounce server.

Countermeasures for Covert Channels

- Defending against covert channel activity can be implemented on the server host and on the network.

- The best defense on the server side is to prevent attackers from gaining access and installing covert channels.

- This is not always a foolproof measure however. Therefore it is important to deploy mechanisms to detect such activity at the network level.

- It is clear that a stateless IDS is unable to detect covert channel activity. Tools such as tcpdump on the other hand are very effective in providing detailed information that can be used to identify and analyze such activity.

- However, this is a very onerous task on busy servers, not to mention the large amounts of storage required.

- A reasonable approach would be to capture traffic from only those addresses that are suspected to be generating anomalous traffic.

- In addition, administrators must be very familiar with processes running on critical systems and periodically inspect the process table for unusual processes.