Linux Raw Socket Programming - What Lies Beneath a Socket?

November 30, 1995, http://www.linuxworld.com/story/34589.htm

Summary
When I was casually examining my server log few months back, I noticed something was going off beam. To my horror, the primary server crashed, unable to take the load. Usually, I don't get that much traffic. Months later, I realized that I was the victim of a DDOS attack. Being a hacker type myself, I tried to investigate where I failed in my system administration.

By Frank Jennings

When I was casually examining my server log few months back, I noticed something was going off beam. To my horror, the primary server crashed, unable to take the load. Usually, I don't get that much traffic. Months later, I realized that I was the victim of a DDOS attack. Being a hacker type myself, I tried to investigate where I failed in my system administration. And I started learning Raw Socket programming, in an effort to understand how powerful it is under Linux. To my surprise, I realized any lamer can build up Raw Socket applications and can effectively misuse this wonderful trait . I'd like to share some of the interesting Raw Socket exploits. Don't ever try this!

All along I was wondering if I could spoof my IP address and perform a SYN Flood attack on a server using C with my Linux box (2.4.1), as I found out that the hackers spoofed their source IP and flooded the server with infinite connection requests. Soon I realized that it is not only possible but also easy to do.

Using Raw Sockets

When you create a socket and bind it to a process/port, you don't care about IP or TCP header fields as long as you are able to communicate with the server. The kernel or the underlying operating system builds the packet including the checksum for your data. Thus, network programming was so easy with the traditional cooked sockets. But it's a good programming practice to create your own TCP/IP header including the checksum and bypass the kernel for many reasons, including better control over your socket and an interesting exploration of what lies beneath a socket. Welcome to the realm of Raw Sockets. Raw Sockets let you fabricate the header fields including information like source IP address. And naturally, this has become a valuable tool for all the lamer kids (they call themselves hackers) in spreading havoc by spoofing IPs. As a programmer, you are entitled to explore this wonderful feature, which lets you define your own networking protocol for specific needs.

In the rest of the article, I will explain Raw Socket creation and ways of exploiting it to create a SYN Flooding machine and a connection termination tool.

IP Spoofing and SYN Flood
Before starting with our connection flooder code, we need to understand the TCP connection process, often termed as a "three-way handshake." The client who needs to initialize a connection sends out a SYN segment (Synchronize) to the server along with the initial sequence number. No data is sent during this process, and the SYN segment contains only TCP Header and IP Header. When the server receives the SYN segment, it acknowledges the request with its own SYN segment, called SYN-ACK segment. When the client receives the SYN-ACK, it sends an ACK for the server's SYN. At this stage the connection is "Established."

The SYN Flooding technique involves spoofing the IP address and sending multiple SYN segments to a server. When the server gets a connection request, it sends a SYN-ACK to the spoofed IP address, which in all probable case 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 finite, flooding the server with continuous SYN segments can slow down the server or completely push it offline. We can also write a code, which sends a SYN packet with a randomly spoofed IP. 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. (They never asked for a connection). This can choke the target server and often form a crucial part of a DDOS attack.

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, introduced recently and now part of most of the Linux kernels, help in completely protecting 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.

If you have the latest kernel and want to enable SYN Cookies, add

```
echo 1 > /proc/sys/net/ipv4/tcp_syncookies
```

to your /etc/rc.d/rc.local script. Edit /etc/sysctl.conf file and add the line:

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

and restart your network. You are now protected against any SYN Flooding.

Building a Spoofed Packet
In Linux, if you want to create a raw socket, you need super user privilege, thus preventing other users from writing their own datagrams to the network. All socket-related structures and functions can be found in sys/socket.h and linux/socket.h. To create an IPV4 raw socket, use the socket function like this:

```
int sd;
sd=socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
```

The protocol constant IPPROTO_TCP is defined in netinet/in.h and linux/in.h. The above code creates an Internet (AF_INET) raw socket with TCP protocol.

Spoofing source IP can be done by setting IP_HDRINCL (Include IP Header) in socket option.

```
int sm=1;
const int *val=&sm;
setsockopt(sd, IPPROTO_IP, IP_HDRINCL, val, sizeof(sm));
```

If the result is a negative number, then the kernel does not allow IP spoofing, which is very rare. With my 2.4.1 kernel, it works. Check out the IP header, which we've built:

```
iph->ip_hl = 5;
iph->ip_v = 4;
iph->ip_tos = 0;
iph->ip_len = sizeof (struct ip) + sizeof (struct tcphdr);
iph->ip_id = htonl (54321);
iph->ip_off = 0;
iph->ip_ttl = 255;
iph->ip_p = 6;
iph->ip_sum = 0; iph->ip_dst.s_addr = sin.sin_addr.s_addr;
tcph->th_sport = htons (3456);
tcph->th_dport = htons (atoi(argv[2]));
tcph->th_seq = random ();
tcph->th_ack = 0;
tcph->th_x2 = 0;
tcph->th_off = 0;
tcph->th_flags = TH_SYN;
tcph->th_win = htonl (65535);
tcph->th_sum = 0;
tcph->th_urp = 0;
iph->ip_sum = csum ((unsigned short *) datagram, iph->ip_len >> 1);
```

Note that we are specifying each and every value in the IP header, including header length, IP version, type of service, and checksum. We'll build a checksum algorithm for our packets later in this article. We have also set the SYN flag for this packet. You can also set ACK flag along with SYN as we have done in our connection terminator example.

Some firewalls, such as ZoneAlarm Pro, detect SYN Flood and block the source IP address (see Figure 1). So we need to assign our spoofed source IP address a random number as shown below:

```
b1=100+(int)(255.0*rand()/(RAND_MAX+100.0));
b2=100+(int)(255.0*rand()/(RAND_MAX+100.0));
b3=100+(int)(255.0*rand()/(RAND_MAX+100.0));
b4=100+(int)(255.0*rand()/(RAND_MAX+100.0)); if(b1>255)
sprintf(b1s,"%d",b1);
sprintf(b2s,"%d",b2);
sprintf(b3s,"%d",b3);
sprintf(b4s,"%d",b4);
strcat(b1s,".");
strcat(b2s,".");
strcat(b3s,".");
strcat(b1s,b2s);
strcat(b1s,b3s);
strcat(b1s,b4s);
iph->ip_src.s_addr = inet_addr (b1s);
```

We must be careful in building the spoofed IP in the correct form, as the kernel is configured to drop packets with malformed addresses. After fabricating our packet we can send the SYN request using the standard sendto() method as follows:

```
sendto (s,datagram,iph->ip_len,0,(struct sockaddr *) &sin,sizeof (sin));
```

Calculating Checksum

If you check out choke.c (available on site), we calculate the checksum for the packet we send using the following

function:

```
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 ~sum;
```

While raw socket offers you more power than the traditional cooked socket, it forces you to calculate the checksum for every data packet you send into the network. The Internet header checksum provides a verification that the information used in processing Internet datagram has been transmitted correctly. If the header checksum fails, the Internet datagram is discarded at once by the entity, which detects the error. Calculating IP and TCP checksum is no big deal as RFC 791 puts in:

'The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header'.

If you can understand the above statement, stop reading this article. For lesser mortals like us, we shall analyze the process of building our checksum algorithm. We calculate checksum only for the header and not for the data, which we send. One very queer thing with IP checksum is that we need to include the value of checksum field in the header for calculating the value of checksum field in the header. Sounds crazy? Yes, it is Internet Protocol. So according to RFC 791, you can use the value 0 for checksum for calculating the final value of checksum. The steps involved in calculating the IP checksum is as follows:

1. First the header is split into a series of 16 bit chunks.
2. All these 16 bit chunks are added to get certain value.
3. Subtract the obtained value from FFFF, which will give you the checksum value.

Calculate IP Checksum
Now let's analyze the IP header values. Check out Figure 2 for IP Header structure. The first entry in the header is the IP version, which is **4** (IPV4). The next entry is the header length, which is **5** for most packets. The next entry is "Type of Service", which is 0. Note that TOS occupies 8 bits of header. So lets calculate our first 16 bit chunk. We know 4 is 100 and 5 is 101 in binary. So our first chunk is:
0100 0101 0000 0000, which is **4500**. Next, 16 bits occupy the total length of the packet. Lets assume the length as 40 bytes (This includes the TCP Header too. The TCP Header's minimum size is 20 bytes). So the second chunk is :
0000 0000 0010 1000, which is 0028. If our Identification field value is 1, then the third chunk becomes:
0000 0000 0000 0001, which is 0001
Fragmentation Offset and flags : 0
which gives the fourth chunk as 0000. Our 8-bit Time to Live value is 255, which is FF. The protocol number is 6 in our case (TCP). So the fifth chunk is FF06. Assume header checksum as 0 so the sixth chunk is 0000. Next field is the source address. So with a source address of 34.26.17.10, we get the seventh and eighth chunk as 221A and 110A. Next is the destination address, which is say 1.2.3.4. We get 0102 and 1304 chunks. By adding up all the chunks, we get 17B59. Now remove the first digit and add it with the rest. i.e 1+7B59 = 7B5A. Subtract the number from FFFF, which gives 84A5, which is our IP Checksum value.

Session Hijacking
Session Hijacking can be done effortlessly with a few lines of code using raw socket. Before proceeding to hijack an open TCP Session, we need to understand the TCP connection termination process. Unlike TCP connection initialization, which is a three-way process, connection termination takes place with the exchange of four-way packets. 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.

As you can see, the connection termination process of TCP is complex, since data integrity is ensured with every packet transferred. Before trying to hijack a TCP connection, we need to understand the TIME_WAIT state. Why should any client be made to wait even after receiving connection termination confirmation from the server? Consider this instance (termed as "incarnation") with 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 we need to set our Network Interface to Promiscuous mode. This can be done by specifying the Socket Type in SocketOpt structure as 'PACKET_MR_PROMISC' e-g sockopt.mr_type= PACKET_MR_PROMISC
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.
3. 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 flag.

An Example
I have built two tools for Linux, one called choke, which SYN floods a target system with randomly spoofed IP address. The other one is a TCP connection terminator, which sets your Ethernet card in Promiscuous mode and sniffs on the local subnet, analyze the sniffed packet, retrieve the source IP, destination IP and ports if the protocol being used is TCP and reset the connection by interchanging the sequence and acknowledgment number. With old kernels writing code to read directly from the Datalink layer was shunned mostly because of the inherent complexity involved.

If you are familiar with an Unix environment, you can use the BSD packet Filter or Data Link provider Interface (DLPI). With the new releases of Linux kernel, you can use an elegant interface called SOCK_PACKET. Explaining BSD Packet Filter and DLPI is beyond the scope of this article so lets deal with Linux specific SOCK_PACKET Interface. Similar to the creation of raw socket, creating SOCK_PACKET socket requires administrative privilege too. You can create a socket like this:

```
sd=socket(AF_INET, SOCK_PACKET, htons(ETH_P_ALL));
```

which will receive all the frames from the datalink layer. Remember you are sniffing a subnet and with high traffic network, the socket will act really strange. You can instead sniff only IP packets using ETH_P_IP frame option. Other options include ETH_P_ARP and ETH_P_IPV6. (Check out linux/if_ether.h). If your kernel and network device supports Promiscuous mode, you can do an ioctl of SIOCGIFLAGS by fetching the flag and setting IFF_PROMISC flag and again storing the flags using SIOCGIFLAGS. Check this from termin.c (Available on site) code:

```
fd = socket(AF_INET, SOCK_PACKET, htons(0x3)))
ioctl(fd,SIOCGIFFLAGS,&ifinfo);
ifinfo.ifr_ifru.ifru_flags |= IFF_PROMISC;
ioctl(fd,SIOCSIFFLAGS,&ifinfo);
```

If you don't know what ioctl does, check out the man page. It lets you manipulate the device parameters of files. And for instructing your network adapter to pass on all the packets it gets to your application, you should set your socket option to SOL_PACKET and bind it to the socket using PACKET_ADD_MEMBERSHIP.

```
sock = socket(PF_PACKET, SOCK_DGRAM, htons(ETH_P_IP));
setsockopt(sock, SOL_PACKET, PACKET_ADD_MEMBERSHIP,
(void *)&sockopt, sizeof(sockopt));
```

The major disadvantage of using SOCK_PACKET is, it does not support kernel filtering and buffering; so multiple frames cannot be passed to our code with a single read method. And also ETH_P_IP returns packets even from a loopback device, so the real challenge lies in discerning and discarding useless frames.

One problem I faced during connection termination was that most of the packets I sniffed came from my loopback device. So every time I try to hijack a connection I had to check if the source and destination IP are same. Then I wait for an ACK flag set packet. This is to ensure that we mess up with only established connections. Once I find an ACK packet I extract the acknowledgment number and use it as the sequence number in my own packet with IP addresses swapped and will send a FIN packet to the server with the acknowledgment number as the summation of original packet sequence number and the data length.

```
sp_seq=pinfo.ack;
sp_ack=pinfo.seq+pinfo.datalen;
```

Since the server gets the correct sequence number, which it is expecting, it will disconnect the connection. Boom!

Broadcast Amplifiers
IP Spoofing has always helped hackers in devising new DDOS attacks. One such technique is to send a large amount of ICMP echo request (Ping) traffic to all known IP broadcast addresses with the spoofed source address of the victim. The router delivering traffic to the broadcast address enables in broadcasting the request, which results in most of the hosts on the network taking the ICMP echo request and replying to it. First it kills the target and next it increases the network traffic. Unfortunately C lets you code one such spoofer for broadcast amplification with ease. Many hacker tools also perform similar function like Smurf.
After creating a raw socket we need to set our socket option to enable us to send broadcast datagrams with spoofed IP in the network. This can be done effortlessly as follows:

```
setsockopt(sd, SOL_SOCKET, SO_BROADCAST, (char *) &bcast, sizeof(bcast));
```

Remember SO_BROADCAST applies only for Datagram packets. Creating the header is just the same as we have done in the previous example. Let us analyze the ICMP related headers in our packet.

```
struct icmp *icmp;
icmp = (struct icmp *) (packet + sizeof(struct ip));
icmp->icmp_type = 8;
icmp->icmp_code = 0;
icmp->icmp_cksum = htons(~(ICMP_ECHO << 8));
```

Note that we are setting our ICMP packet type as 8, which represents Echo request. (Check out RFC 1700 for other types). The ICMP code stands for the nature of error in the communication. Let us make it 0, which stands for 'Net Unreachable' (Who cares for error messages). After construction the echo reply, just send it to any valid broadcast address, or your subnet broadcast address (Say 234.134.255.255), and check out the burst in network traffic.

Conclusion
I am not a hacker. Nor am I a C programming expert. I have just pointed out few flaws in our networking and communication architecture, which the hackers have exploited all these years. Few years back Yahoo was down for 3 hours-a victim of Spoofing with broadcast amplifiers. With ARP Spoofing already getting popular and hackers exploring possibilities of exploits for undetectable DDOS attack, we can expect more network chaos in the future. If you still swear on TCP/IP as the best communication protocol, tell me, why?

About the author
Frank Jennings works in the Communication Designs Group of Pramati Technologies

Related Sites
· Source Code