Networking Lab: TCP/IP Packet Filtering

1 Introduction

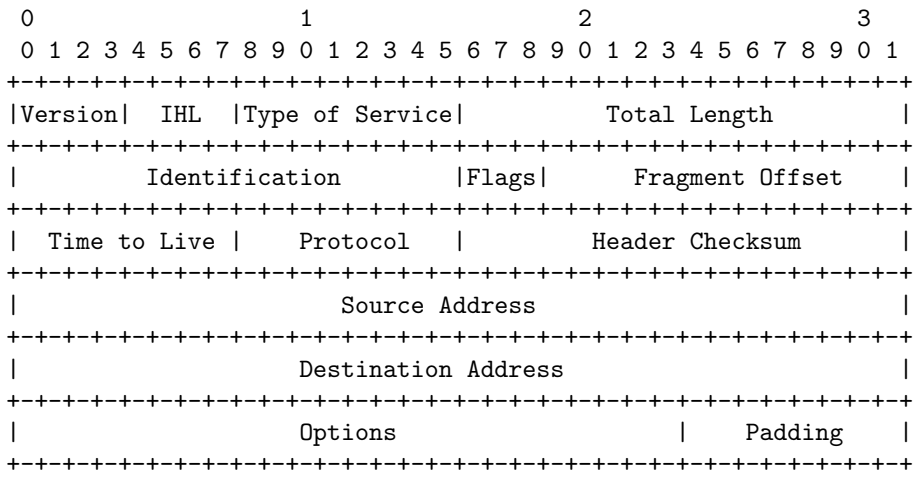
Packet filtering is a technique that can be used for many tasks. It can be used for techniques such as port forwarding, network address translation, packet inspection and packet rewriting. In this assignment, you will implement your own packet filter running on Linux. Under Linux, a mechanism has been implemented to enable packet inspection and rewriting in user space. The kernel running on our system support this mechanism. For user space packet rewriting, we will use the library *libnetfilter_queue*[1] Using this library, you will implement a basic form of encryption, which is completely transparent to applications that use TCP/IP. Eventually, you will have to connect to web server that is running an implementation of this encryption algorithm.

First, a short overview of TCP/IP is given. Next, the tools you will need in this assignment are discussed. The tools are: iptables and libnetfilter_queue. Then the specification of the encryption scheme is given which you should implement using the tools mentioned above.

2 TCP/IP

TCP (Transmission Control Protocol) is a connection oriented protocol that is used for most communication on computer networks nowadays. TCP runs on top of IP (Internet Protocol). IP is used in host-to-host communication in packet-switched networks. It sends data packets called datagrams from one host to another hosts, based on the target address specified. Note that the familiar port numbers (such as 80 for HTTP and 25 for SMTP) are not part of the IP protocol. Multiplexing is handled in the TCP layer. We show the layout of the IP header here (from RFC 791):

[1] http://www.netfilter.org/projects/libnetfilter_queue/index.html



Example Internet Datagram Header

For a description of all these field, please consult RFC 791[2]. Under Linux, the header file /usr/include/linux/ip.h contains the definition of the structure *iphdr*, which looks as follows:

```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
    __u8    ihl:4,
        version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
    __u8    version:4,
        ihl:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
    __u8    tos;
    __be16  tot_len;
    __be16  id;
    __be16  frag_off;
    __u8    ttl;
    __u8    protocol;
    __u16   check;
    __be32  saddr;
    __be32  daddr;
    /*The options start here. */
};
```

Note that these structures are defined in such a way that they will be compatible with both little endian and big endian architectures. Be aware of the fact that

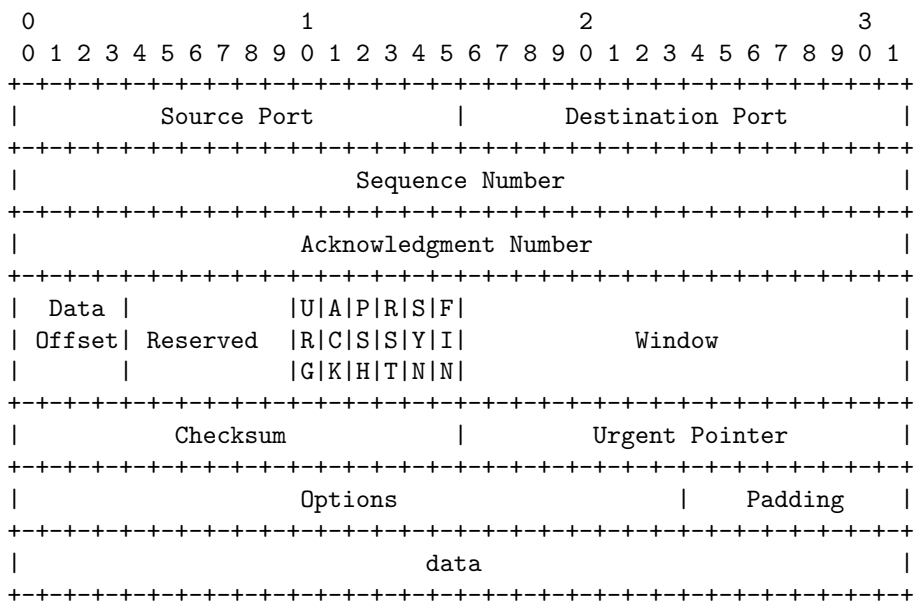
[2]http://www.ietf.org/rfc/rfc0791.txt

all data in the IP header is big endian. GCC supports C bitfields. The notation "__u8 ihl:4;" declares a 4-bit unsigned integer. While the order of definition matters, no conversion needs to be done for values of 8 bit and smaller (thus, endianness only matters at the byte level, not the bit level).

For larger fields (16 bit and greater), the network byte order must be converted to the host byte order, if they are to be used in integer computations. This conversion is architecture dependent and you should use the functions *ntohs, ntohl, htons* and *htonl* whenever appropriate.

If you want to access fields of an IP header, you should not define your own structure, but use the one provided by the header file mentioned above.

The data payload of an IP packet is located directly after the header. Its size can be determined using the IP header. In our case the payload will be a TCP packet. The layout of such a packet is as follows (RFC 793[3]):



TCP Header Format

The corresponding structure is defined in /usr/include/linux/tcp.h:

[3]http://www.ietf.org/rfc/rfc0793.txt

```
struct tcphdr {
    __u16   source;
    __u16   dest;
    __u32   seq;
    __u32   ack_seq;
#if defined(__LITTLE_ENDIAN_BITFIELD)
    __u16   res1:4,
        doff:4,
        fin:1,
        syn:1,
        rst:1,
        psh:1,
        ack:1,
        urg:1,
        ece:1,
        cwr:1;
#elif defined(__BIG_ENDIAN_BITFIELD)
    __u16   doff:4,
        res1:4,
        cwr:1,
        ece:1,
        urg:1,
        ack:1,
        psh:1,
        rst:1,
        syn:1,
        fin:1;
#else
#error  "Adjust your <asm/byteorder.h> defines"
#endif
    __u16   window;
    __u16   check;
    __u16   urg_ptr;
};
```

All fields are described in RFC 793. The field *check* is the checksum. Checksum recomputation is essential if you rewrite a TCP packet. While checksum computation is described in RFC 793 this can be a bit tricky and therefore we outline the checksum computation here. For checksum computation purposes, the TCP packet is prepended with a pseudo header. While this pseudo header does not really have to be constructed, you should view it as being right in front of a TCP packet when computing the checksum.

```
+--------+--------+--------+--------+
|           Source Address          |
+--------+--------+--------+--------+
|         Destination Address       |
+--------+--------+--------+--------+
|  zero  |  PTCL  |    TCP Length   |
+--------+--------+--------+--------+
```

The source and destination address and the protocol can be obtained from the IP packet. TCP length is the TCP header length plus the length of the data payload. This quantity must be computed.

Using the pseudo header prepended to the TCP packet, the checksum is computed as follows (See RFC 1072[4] for more details on byte order independence):

```
long checksum( unsigned short *addr, int count )
{
    /* Compute Internet Checksum for "count" bytes
     *         beginning at location "addr".
    */
    long checksum;
    register long sum = 0;

    while( count > 1 )  {
        /*  This is the inner loop */
        sum += * (unsigned short) addr++;
        count -= 2;
    }

    /*  Add left-over byte, if any */
    if( count > 0 )
        sum += * (unsigned char *) addr;

    /*  Fold 32-bit sum to 16 bits */
    while (sum>>16)
        sum = (sum & 0xffff) + (sum >> 16);

    checksum = ~sum;
    return checksum;
}
```

3 Programming Using Sockets

In order to test your packet encryption implementation later on, you should implement a simple TCP server that only echos data received. A client (you can just use *telnet*) should connect to this server and send a test string, which

[4] http://tools.ietf.org/html/rfc1071

must in turn be received back from the server. In order to create a TCP server, the following calls are needed in this order (use the man pages as a reference):

1. socket
2. bind
3. listen
4. accept
5. recv
6. send
7. close

4 Packet Filtering Using Iptables

We will use the following settings using iptables:

```
iptables -t mangle -A INPUT -j NFQUEUE --queue-num 0
iptables -t mangle -A OUTPUT -j NFQUEUE --queue-num 1
```

Read the man page of iptables (*man iptables*) and understand what these commands do. If these rules are in effect, you network will not be functioning unless you handle the packets that are sent to user space. To flush all rules from iptables, use:

```
iptables -F
iptables -t mangle -F
```

5 User Space Packet Filtering Using Libnetfilter_queue

Libnetfilter_queue is a library for packet filtering in user space. Version 0.0.13 is installed on the machines you will be using. You can download the sources yourself from www.netfilter.org. This includes the file *nfqnl_test.c*, which shows how to handle packets with this library.

6 Packet Level Encryption

It is your task to implement a 32-bit XOR based payload encryption. All implementations among the groups should be compatible with each other, i.e. you should be able to communicate with other groups provided that the 32-bit key is given. A reference machine is available to test your implementation. The key of the reference implementation is 0x12345678. This key should be stored

BIG ENDIAN, as all network traffic is big endian. If the payload is not a multiple of 32 bits, then only part of the key is used for the last few bytes. Example:

```
DATA: AB CD EF 34 23 21 34  (7 bytes)
KEY : 12 34 56 78 12 34 56  (key repeated each 4 bytes)
```

7 Requirements

Implement the echo server. Your source code should be readable and contain useful comments. No further documentation is required for the echo server.

For the TCP payload encryption program, you must write readable code. You must implement the following functions:

- struct iphdr *get_ip_hdr(char *data)
- struct tcphdr *get_tcp_hdr(char *data)
- char *get_tcp_payload(char *data)
- __u16 get_ip_tot_len(struct iphdr *ip)
- __u16 get_tcp_opt_len(struct tcphdr *hdr)
- __u16 get_tcp_data_len(char *data)
- __u16 get_tcp_source(struct tcphdr *hdr)
- __u16 get_tcp_dest(struct tcphdr *hdr)
- void encrypt_payload(char *data)

All functions taking "char *data" get the full packet including IP header. Encrypt payload should rewrite the TCP payload of the IP packet given by *data*. Your implementation must be able to setup an encrypted connected to one of our web servers.