**2.5:**What's wrong with the protocol in §2.4.4 *Authentication*? (Hint: assume Alice can open two connections to Bob.)

Answer: This is the same as the reflection attack on the Chapter 11. If there is an imposter Cathy between Alice and Bob, firstly, Cathy imposts Alice to communicate with Bob, for Bob can hear but can not see him. Cathy uses a random challenge rc to Bob. And Bob replies encrypted ra with Kab and his own challenge rb.

Session 1:

Cathy -----------------(rc)-------🡪 Bob

Cathy 🡨--------------(rb)---------- Bob

Cathy 🡨----(encrypted(ra))----- Bob

Next, Cathy starts another session 2 with Bob, however, this time, he uses the challenge rb that he received from last session. Cathy does the same thing to Bob. Bob replies encrypted rb with Kab and his NEW challenge rx. However, Cathy does not care about rx at all.

Session 2:

Cathy -----------------(rb)-------🡪 Bob

Cathy 🡨--------------(rx)---------- Bob

Cathy 🡨----(encrypted(rb))----- Bob

Now, Cathy gets the encrypted(rb). He can do two things. One, Cathy can use the second encrypted(rb) directly to reply the session 1 to complete the initiation. Another thing is that he can decrypt the the encrypted function and Kab with statistical analysis.

**3.3:**How many DES keys, on the average, encrypt a particular plaintext block to a particular ciphertext block?

Answer: There are many DES types, such as, DES-64, DES-128 and so on. Assume that the plaintext and the ciphertext of the DES is k bits and the key for the DES algorithm is m bits. According to the definition of DES, the m bits of keys includes m/8 bits for sanity-check of each 8 bits. Therefore, the 2k of plaintext block map to the 2k of ciphertext with 2m-m/8 combination of keys. On the average, each plaintext block has 2m-m/8/2k keys. As we can see, the DES-64 has 264-64/8/264 = 1/28 keys on average. The DES-128 has 2128-128/8/2128 = 1/216 keys on average.

3. Let *Ln*, *Rn*, *Kn* denote 32-bit, 32-bit and 48-bit random numbers respectively, and let *Re*(*Ln*, *Rn*, *Kn*) = (*Ln*+1, *Rn*+1) represent the DES encryption round shown in the left diagram in Figure 3.6 in the textbook, which has included 2 functional mappings:

* 1. *Ln*×*Rn*×*Kn* ⇒ *Ln*+1 (this means *Ln*+1 is a function of *Ln*, *Rn*, *Kn*)
  2. *Ln*×*Rn*×*Kn* ⇒ *Rn*+1 (this means *Rn*+1 is a function of *Ln*, *Rn*, *Kn*)

1) Prove that *Re*(*Rn*+1, *Ln*+1, *Kn*) = (*Rn*, *Ln*)

Answer: According to the definition of DES, assume F(x, k) is the Mangler Function

*Ln*×*Rn*×*Kn* ⇒ *Ln*+1: Ln+1 = Rn and

*Ln*×*Rn*×*Kn* ⇒ *Rn*+1: Rn+1 = Ln xor F(Rn, Kn)

*Re*(*Rn*+1, *Ln*+1, *Kn*) = (*Rn+2*, *Ln+2*) has two mappings:

*Rn+1*×*Ln+1*×*Kn* ⇒ *Rn*+2: Rn+2 = Ln+1 and

*Rn+1*×*Ln+1*×*Kn* ⇒ *Ln*+2: Ln+2 = Rn+1 xor F(Ln+1, Kn)

As we know, Ln+1 = Rn , then F(Rn, Kn) = F(Ln+1, Kn)

So Ln+2 = Rn+1 xor F(Ln+1, Kn)

= [Ln xor F(Rn, Kn)] xor F(Rn, Kn)

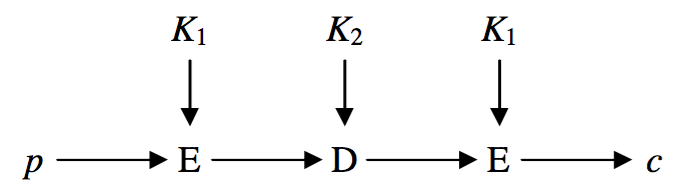
= Ln

Rn+2 = Ln+1 = Rn

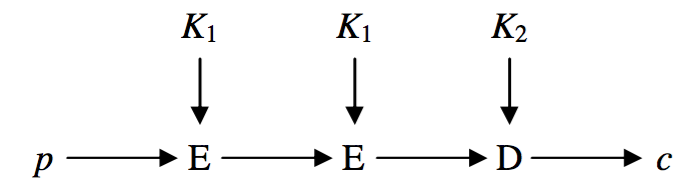
As a result: *Re*(*Rn*+1, *Ln*+1, *Kn*) = (*Rn*, *Ln*).

4. In triple DES (represented as 3DES), two 56-bit random keys (*K*1 & *K*2) are used in the

following way:



We change the order of the three DES operations in 3DES to the following:



and call it 3DES’. We measure the strength of 3DES and 3DES’ by the effort (in term of number of tries) in

finding some *K*1 & *K*2 to match a known plaintext and ciphertext pair <*p*, *c*>.

1)  Does 3DES’ have the same strength as 3DES?

2)  Why?

Answer: (1) No, the 3DES’ weakens the strength of the encryption.

(2) There are two reasons for the 3DES’ weaker than the 3DES. First, the two K1 encryption equals to one K3 encryption. The hacker can replace the two K1 encryption with another K3 encryption which produce the same result. Because the DES algorithm adopts the XOR operation to the plaintext. Second, the 3DES’ creates the condition for meet-in-the-middle attack between K1 and K2 encryption. The operation from p to c must go through the encryption and decryption of K1 and K2. And the decipher from c to p also go through encryption and decryption of K2 and k1. The DES algorithm is reversible. Therefore, the encryption and decipher generates the same middle message between the two K1 and K2. An unauthorized user may emulate both K1 and K2 to see whether they are match in the middle between two K1 and K2. Even though, the unauthorized user does not have the same K1 and K2. He can find another K3 and K4 that have the same effect as two K1 and K2. This 3DES’ extremely reduce the strength of the 3DES encryption. Meet-in-the middle attack will not work on the 3DES encryption, because there is no match in the middle of the encryption.

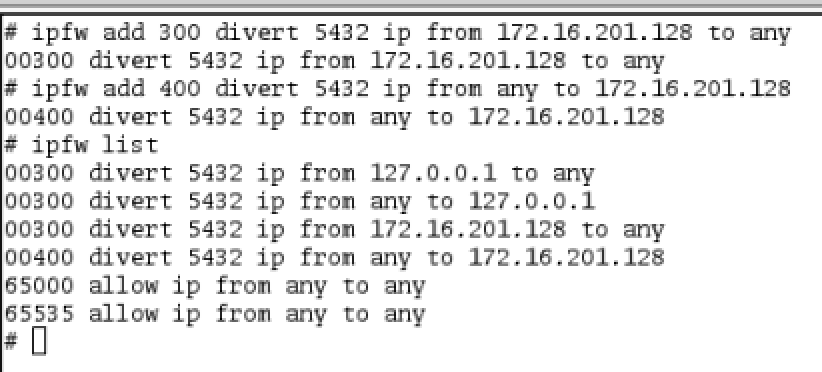
5. Implement toy firewall via divert socket

1) You are required to write a (could be C or C++) program called block\_allICMP that uses the divert docket in the FreeBSD 5.4 VM to block all incoming ICMP packets to the FreeBSD 5.4 VM.

Answer: The FreeBSD uses the *ipfw* to redirect ICMP packets to a certain socket port. Then write a program to manage all the packets that redirected by the *ipfw*.

To make sure ipfw does not block any packets type command to the terminal:

ipfw add 65000 allow ip from any to any



The controlling program was using the divert socket, which is a kind of raw socket but does not require IP address in the bind() function.

/\* divertlib.h

convenient library header file for FreeBSD divert socket

Ruikang Dai, 10/04/2015

\*/

#include <sys/time.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netinet/in\_systm.h>

#include <netinet/ip.h>

#include <netinet/tcp.h>

#include <stdio.h>

int initDivSock (u\_short divport);

int readDiv(int sockd, u\_char \*buf, int buflen, struct sockaddr \*saddr);

int writeDiv(int sockd, u\_char \*buf, int buflen, struct sockaddr \*saddr);

/\* block\_allICMP.c

loop of FreeBSD divert socket

Ruikang Dai 10/4/2015

\*/

#define DEBUG

#include "divertlib.h"

#define maxbuflen 1520

int main(int argc, char \*argv[])

{ int i, len, divsock;

u\_short iphlen, tcphlen;

int udpsock, DivPort;

struct sockaddr\_in sin, sin1;

struct ip \*iph;

struct tcphdr \*tcph;

struct udphdr \*udph;

unsigned char buf[maxbuflen+1];

int addrsize=sizeof (struct sockaddr);

if (argc!=2)

{ puts("usage : divert-loop [div port]");

return 1;

};

DivPort=atoi(argv[1]);

printf("DivPort=%d\n", DivPort);

if ((divsock=initDivSock(DivPort))<=0)

{ printf("can not get divert socket for port %d, divsock=%d\n",

DivPort, divsock);

exit(1);

};

for (i=1; ;i++)

{

/\* if ((len=readDiv(divsock, buf, maxbuflen, (struct sockaddr \*) &sin))>0)

\*/

if ((len=recvfrom(divsock, buf, maxbuflen, 0,

(struct sockaddr \*) &sin, &addrsize))>0)

{

iph=(struct ip \*) buf;

iphlen=iph->ip\_hl<<2;

#ifdef DEBUG

if (sin.sin\_addr.s\_addr==INADDR\_ANY) /\* outgoing \*/

{ printf("\n%d : Out\t\t\t\t\t\t\t\t==>\n", i);

}

else /\* incoming \*/

{ printf("\n%d : In from %s:%d\t\t\t\t\t\t<==\n",

i, inet\_ntoa(sin.sin\_addr), ntohs(sin.sin\_port));

};

printf("\tsrc IP:%s\n", inet\_ntoa(iph->ip\_src));

printf("\tdst IP:%s\n", inet\_ntoa(iph->ip\_dst));

printf("\tproto :%d\n", iph->ip\_p);

#endif

printf("%d", iph->ip\_p);

if (1 != iph->ip\_p) {

sendto(divsock, buf, len, 0, (struct sockaddr \*) &sin,

sizeof(struct sockaddr));

}

else {

printf("ICMP packets received!\n");

}

}

};

return 0;

}

/\* divertlib.c

convenient library of FreeBSD divert socket

Ruikang Dai 10/4/2015

\*/

#include "divertlib.h"

int initDivSock (u\_short divport)

/\* return >0 : Ok

-1 : error

\*/

{

struct sockaddr\_in addr;

int sockfd;

if (divport)

{ sockfd = socket(PF\_INET, SOCK\_RAW, IPPROTO\_DIVERT);

if (sockfd == -1)

{

#ifdef DEBUG

fprintf(stderr, "Unable to create divert socket\n");

#endif /\* DEBUG \*/

puts("unable to create divert socket");

return -1;

}

/\* bind socket \*/

addr.sin\_family = AF\_INET;

addr.sin\_addr.s\_addr= INADDR\_ANY;

addr.sin\_port = htons(divport);

if (bind( sockfd,

(struct sockaddr \*) &addr,

sizeof addr) == -1)

{

#ifdef DEBUG

fprintf(stderr, "Unable to bind divert socket : %d\n", divport);

#endif /\* DEBUG \*/

puts("unable to bind divert socket");

return -1;

}

return sockfd;

}

else return -1;

}

int readDiv(int sockd, u\_char \*buf, int buflen, struct sockaddr \*saddr)

/\* return >0 : number of bytes read

else : error

\*/

{ int len;

int addrsize=sizeof (struct sockaddr);

len = recvfrom(sockd, buf, buflen, 0, saddr, &addrsize);

return len;

}

int writeDiv(int sockd, u\_char \*buf, int buflen, struct sockaddr \*saddr)

/\* return actural number of bytes writen

\*/

{ int len;

len = sendto(sockd, buf, buflen, 0, saddr, sizeof (struct sockaddr));

return len;

}

The critical part of this program is in the block\_allICMP. The iph->ip\_p is the type of protocol that used in the packet. If iph->ip\_p equals to 1, the packet is a ICMP packet. I added a if condition to ignore the ICMP packets:

if (1 != iph->ip\_p) {

sendto(divsock, buf, len, 0, (struct sockaddr \*) &sin, sizeof(struct sockaddr));

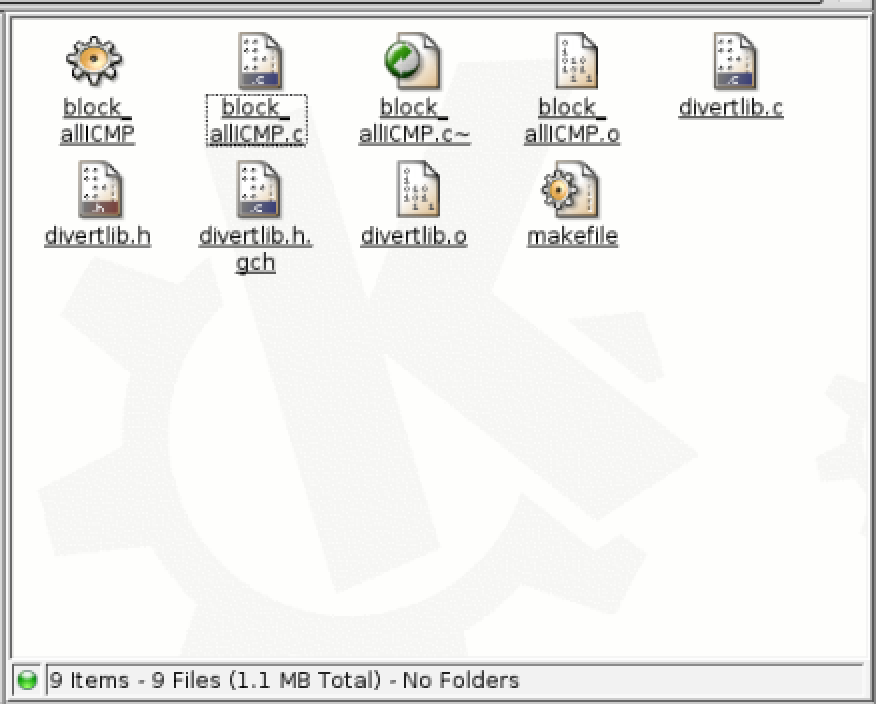
}

Compile these three file with gcc:

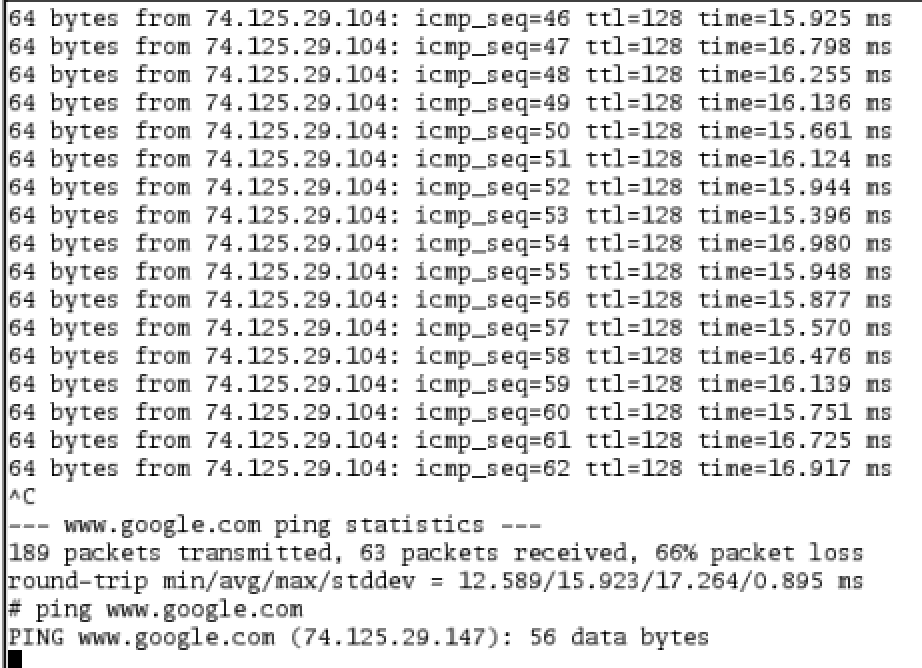
gcc -c divertlib.c divertlib.h -o divertlib.o

gcc -c block\_allICMP.c

gcc block\_allICMP.o divertlib.o -o block\_allICMP



The “ping” command is still working

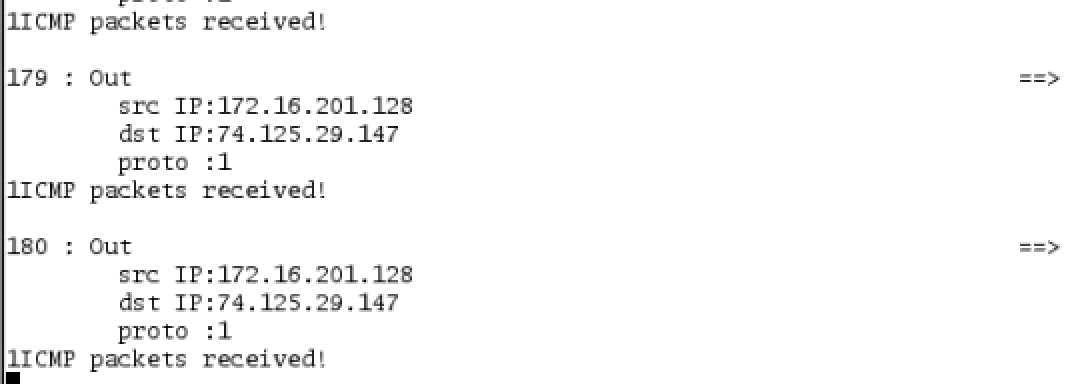


When the program starts and I added divert rules to ipfw:

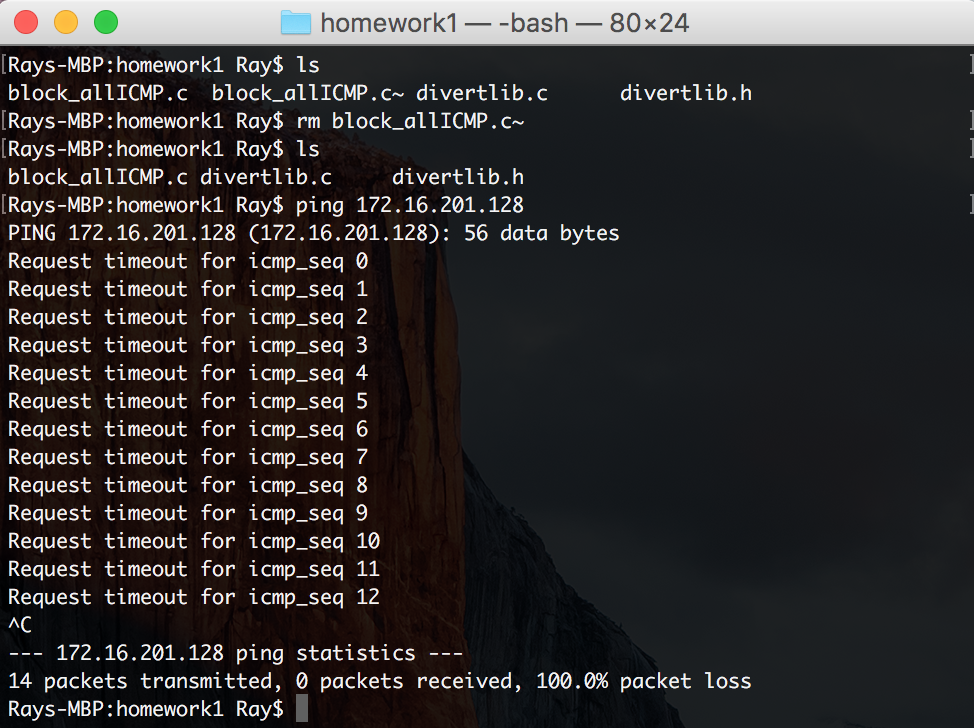
ipfw add 300 divert 5432 ip from 172.16.201.128 to any

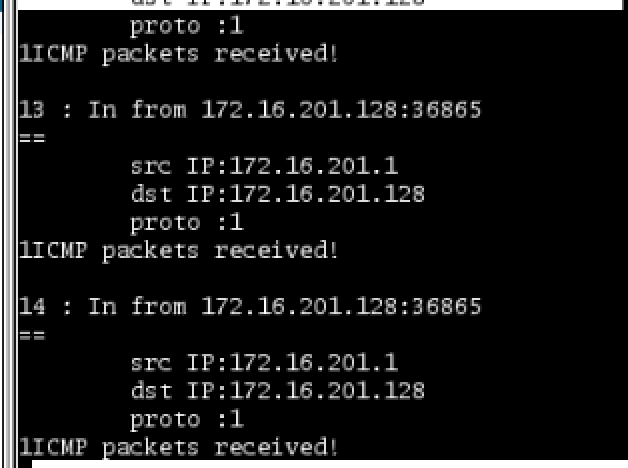
ipfw add 400 divert 5432 ip from any to 172.16.201.128

The “ping” cannot send any packets out to the network anymore, all the ICMP packets can find in the “block\_allICMP” program:



The packets from outsize:





2)  You are required to write a (could be C or C++) program called block\_inICMP that uses the divert docket in the FreeBSD 5.4 VM to block all incoming ICMP packets but allow incoming ICMP echo reply packets to the FreeBSD 5.4 VM

Answer: Nothing have to be change from previous program. The only file I changed is the block\_allICMP.c file. Then I can get another block\_inICMP.c file that block only the in coming ICMP request. The most important part is the else part that I have not implemented:

if (1 != iph->ip\_p) {

sendto(divsock, buf, len, 0, (struct sockaddr \*) &sin, sizeof(struct sockaddr));

}

I added the go through rule in the else part:

else {

// ICMP request to destination

if (inet\_addr("172.16.201.128") == iph->ip\_src.s\_addr && 8 == buf[(iph->ip\_hl)\*4])

sendto(divsock, buf, len, 0, (struct sockaddr \*) &sin,

sizeof(struct sockaddr));

// ICMP reply to local

if (inet\_addr("172.16.201.128") == iph->ip\_dst.s\_addr && 0 == buf[(iph->ip\_hl)\*4])

sendto(divsock, buf, len, 0, (struct sockaddr \*) &sin,

sizeof(struct sockaddr));

}

In these two condition buf[(iph->ip\_hl)\*4] is the type of ICMP type. If buf[(iph->ip\_hl)\*4] equals to 8, the packet is requesting the ICMP information from destination. If buf[(iph->ip\_hl)\*4] equals 0, the packet is replying the ICMP information from the destination. And all the packets from local host out going request and all the packets from outside host can go through the divert socket program. Here are the result of the block\_inICMP program:

