网络安全技术 ——— 端口扫描器的设计与实现

> **学号：2111033**
> **姓名： 艾明旭 专业：信息安全**

一、实验目的

端口扫描器是一种重要的网络安全检测工具。通过端口扫描，不仅可以发现目标主机的开放端口和操作系统的类型，还可以查找系统的安全漏洞，获得弱口令等相关信息。因此，端口扫描技术是网络安全的基本技术之一，对于维护系统的安全性有着十分重要的意义。

本章编程训练的目的如下： ① 掌握端口扫描器的基本设计方法。 ② 理解 ping 程序，TCP connect 扫描，TCP SYN 扫描，TCP FIN 扫描以及 UDP 扫描的 工作原理。 ③ 熟练掌握 Linux 环境下的套接字编程技术。 ④ 掌握 Linux 环境下多线程编程的基本方法

二、实验内容

本章编程训练的要求如下： ① 编写端口扫描程序，提供 TCP connect 扫描，TCP SYN 扫描，TCP FIN 扫描以及 UDP扫描 4 种基本扫描方式。 ② 设计并实现 ping 程序，探测目标主机是否可达。

三、实验步骤

1、相关概念介绍

> ping基本概念

ping 程序是日常网络管理中经常使用的程序。它用于确定本地主机与网络中其它主机的通信情况。因为只是简单地探测某一 IP 地址所对应的主机是否存在，因此它的原理十分简单。扫描发起主机向目标主机发送一个要求回显（type = 8，即为 ICMP_ECHO）的 ICMP数据包，目标主机在收到请求后，会返回一个回显（type = 0，即为 ICMP_ECHOREPLY）的 ICMP 数据包。扫描发起主机可以通过是否接收到响应的 ICMP 数据包来判断目标主机是否存在。 在本章编程中，可以在向目标主机发起端口扫描之前使用 ping 程序确定目标主机是否存在。如果 ping 目标主机成功，则继续后面的扫描工作；否则，放弃对目标主机的扫描。

> TCP扫描

TCP连接建立过程：

TCP 协议的包头结构如图 2-3 所示。对于 TCP 端口扫描而言，TCP 协议的标志位起着至关重要的作用。各标志位的含义参考本书第 2 章关于 TCP 协议的介绍，此处不再赘述。虽然 TCP 扫描的种类繁多，但是它们的原理都与 TCP 连接的建立过程密切相关。在介绍各种 TCP 扫描之前，必须深入理解一个 TCP 连接的建立过程。该过程的步骤如下：(1) 首先客户端（请求方）在连接请求中，向服务器端（接收方）发送 SYN=1，ACK=0的 TCP 数据包，表示希望与服务器建立一个连接。(2) 如果服务器端响应这个连接请求，就返回一个SYN=1，ACK=1的数据包给客户端，表示服务器端同意建立这个连接，并要求客户端进行确认。(3) 最后客户端发送一个 SYN=0，ACK=1的数据包给服务器端，表示确认建立连接

> TCP协议相关的三种扫描

（1）connect扫描

TCP connect 扫描非常简单。扫描发起主机只需要调用系统 API connect 尝试连接目标主机的指定端口，如果 connect 成功，意味着扫描发起主机与目标主机之间至少经历了一次完整的 TCP 三次握手建立连接过程，被测端口开放；否则，端口关闭。

虽然在编程时不需要程序员手动构造 TCP 数据包，但是 connect 扫描的效率非常低下。由于 TCP 协议是可靠协议，connect 系统调用不会在尝试发送第一个 SYN 包未得到响应的情况下就放弃，而是会经过多次尝试后才彻底放弃，因此需要较长的时间。此外，connect失败会在系统中造成大量连接失败日志，容易被管理员发现。

（2）SYN

TCP SYN 扫描是使用最广泛的扫描方式，其原理就是向待扫描端口发送 SYN 数据包。如果扫描发起主机能够收到 ACK|SYN 数据包，则表示端口开放；如果收到 RST 数据包，则表示端口关闭。如果未收到任何数据包，且确定目标主机存在，那么发送给被测端口的SYN 数据包可能被防火墙等安全设备过滤。因为 SYN 扫描不需要完成 TCP 连接的三次握手过程，所以它又被称为半开放扫描。

SYN 扫描的最大优点就是速度快。在 Internet 中，如果不考虑防火墙的影响，SYN 扫描每秒钟可以扫描数千个端口。但是由于其扫描行为较为明显，SYN 扫描容易被入侵检测系统发现，也容易被防火墙屏蔽。同时构造原始的 TCP 数据包也需要较高的系统权限（在Linux 中仅限于 root 账户）。

(3) FIN 扫描

TCP FIN 扫描会向目标主机的被测端口发送一个 FIN 数据包。如果目标主机没有任何响应且确定该主机存在，那么表示目标主机正在监听这个端口，端口是开放的；如果目标主机返回一个 RST 数据包且确定该主机存在，那么表示目标主机没有监听这个端口，端口是关闭的。

FIN 扫描具有良好的隐蔽性，不会留下日志。但是它的应用具有很大的局限性，由于不同系统实现网络协议栈的细节不同，FIN 扫描只能扫描 Linux/UNIX 系统。对于 Windows 系统而言，由于

无论端口开放与否，都会返回 RST 数据包，因此对端口的状态无法进行判断。

> UDP 扫描

一般情况下，当向一个关闭的 UDP 端口发送数据时，目标主机会返回一个 ICMP 不可

达（ICMP port unreachable）的错误。UDP 扫描就是利用了上述原理，向被扫描端口发送 0字节的 UDP 数据包，如果收到一个 ICMP 不可达响应，那么就认为端口是关闭的；而对于那些长时间没有响应的端口，则认为是开放的。

但是，因为大部分系统都限制了 ICMP 差错报文的产生速度，所以针对特定主机的大范围 UDP 端口扫描的速度非常缓慢。此外，UDP 协议和 ICMP 协议是不可靠协议，没有收到响应的情况也可能是由于数据包丢失造成的，因此扫描程序必须对同一端口进行多次尝试后才能得出正确的结论。

2、重要函数分析

此次实验主要涉及五个文件，分别是：main.cpp , TCPConnectScan.cpp , TCPFINScan.cpp , TCPSYNScan.cpp , UDPScan.cpp , header.h

IP 头、TCP 头和 TCP 伪头以及一些工具函数

TCP 头，用于发送 TCP 报文

TCP 伪头，用于计算 TCP 头的校验和

```
struct TCPConHostThrParam
{    std::string HostIP;    unsigned HostPort;};

struct TCPConThrParam
{    std::string HostIP;    unsigned BeginPort;    unsigned EndPort;};

struct pseudohdr
{      unsigned int saddr;     unsigned int daddr;     char useless;
    unsigned char protocol;     unsigned short length; };

struct TCPSYNHostThrParam
{    std::string HostIP;    unsigned HostPort;
    unsigned LocalPort;    unsigned LocalHostIP;};

struct TCPSYNThrParam
{    std::string HostIP;    unsigned BeginPort;
    unsigned EndPort;    unsigned LocalHostIP;};


struct TCPFINThrParam
{    std::string HostIP;    unsigned BeginPort;
    unsigned EndPort;    unsigned LocalHostIP;};

struct UDPThrParam
{    std::string HostIP;    unsigned BeginPort;
    unsigned EndPort;    unsigned LocalHostIP;};

struct UDPScanHostThrParam
{    std::string HostIP;    unsigned HostPort;
    unsigned LocalPort;    unsigned LocalHostIP;};

struct ipicmphdr
{     struct iphdr ip;     struct icmphdr icmp; };

struct TCPFINHostThrParam
{    std::string HostIP;    unsigned HostPort;
    unsigned LocalPort;    unsigned LocalHostIP;};

struct TCPHeader {    uint16_t srcPort;   uint16_t dstPort;
  uint32_t seq;    uint32_t ack;    uint8_t null1 : 4;
    uint8_t length : 4;    uint8_t FIN : 1;    uint8_t SYN : 1;
    uint8_t RST : 1;    uint8_t PSH : 1;    uint8_t ACK : 1;
    uint8_t URG : 1;    uint8_t null2 : 2;    uint16_t windowSize;
    uint16_t checkSum;    uint16_t ptr;
};
```

IP 头，用于发送 IP 报文

```
struct IPHeader {    unsigned char headerLen : 4;
    unsigned char version : 4;    unsigned char tos;
    unsigned short length;    unsigned short ident;
    unsigned short fragFlags;    unsigned char ttl;
    unsigned char protocol;    unsigned short checksum;
    unsigned int srcIP;    unsigned int dstIP;
    IPHeader(unsigned int src, unsigned int dst, int protocol) {
        version = 4;        headerLen = 5;        srcIP = src;
        dstIP = dst;    ttl = (char)128;    this -> protocol = protocol;
        if (protocol == IPPROTO_TCP) {   length = htons(20 + 20);  }
        else if (protocol == IPPROTO_UDP) {length = htons(20 + 8);}
    }};
```

校验和计算函数

```
static inline unsigned short in_cksum(unsigned short *ptr, int nbytes)
{
    register long sum;     u_short oddbyte;
    register u_short answer;     sum = 0;
    while(nbytes > 1)
    { sum += *ptr++; nbytes -= 2;     }

    if(nbytes == 1)
    {         oddbyte = 0; *((u_char *) &oddbyte) = *(u_char *)ptr;
        sum += oddbyte;     }

    sum = (sum >> 16) + (sum & 0xffff); sum += (sum >> 16);
    answer = ~sum;     return(answer); }
```

获取本地 IP 地址

```
static inline unsigned int GetLocalHostIP(void)
{     FILE *fd;     char buf[20] = {0x00};
    fd = popen("/sbin/ifconfig | grep inet | grep -v 127 | awk '{print $2}' | cut -d
\":\" -f 2", "r");
    if(fd == NULL)
    { fprintf(stderr, "cannot get source ip -> use the -f option\n");
        exit(-1);     }
    fscanf(fd, "%20s", buf);     return(inet_addr(buf)); }
```

ICMP 探测指定主机

测量本地主机与目标主机之间的网络通信情况，用 ping 函数实现。具体实现为首先我们需要建立一个套接字用来通信，并设置我们需要发送的 IP 包。然后我们创建 ICMP 请求数据包，并对 ip 头和 icmp 头进行填充，为了保证对面成功接受并进行应答。然后我们设置套接字的发送地址，

即我们需要扫描的目标地址，并向目标地址发送我们的 icmp的数据包。然后循环接受 icmp 响应，具体为首先获得循环起始时间，然后创建一个 ICMP 接受数据包，进入循环，如果接收到一个数据包则对其进行解析，获得响应数据包的 IP 头的原地址、目的地址，然后判断该数据包的源地址是否等于被测主机的 IP 地址和目的地址是否相等 ICMP 头的 type字段是否为 ICMP_ECHOREPLY，如果等待时间超过三秒则是失败。

```
bool Ping(std::string HostIP, unsigned LocalHostIP) {
    struct iphdr *ip;
    struct icmphdr *icmp;
    unsigned short LocalPort = 8888;

    int PingSock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

    if(PingSock < 0) {
        std::cout << "socket error" << std::endl;
        return false;
    }

    int on = 1;
    int ret = setsockopt(PingSock, 0, IP_HDRINCL, &on, sizeof(on));

    if(ret < 0) {
        std::cout << "setsockopt error" << std::endl;
        return false;
    }

    int SendBufSize = sizeof(struct iphdr) + sizeof(struct icmphdr) + sizeof(struct
timeval);
    char *SendBuf = (char*)malloc(SendBufSize);
    memset(SendBuf, 0, sizeof(SendBuf));

    ip = (struct iphdr*)SendBuf;
    ip -> ihl = 5;
    ip -> version = 4;
    ip -> tos = 0;
    ip -> tot_len = htons(SendBufSize);
    ip -> id = rand();
    ip -> ttl = 64;
    ip -> frag_off = 0x40;
    ip -> protocol = IPPROTO_ICMP;
    ip -> check = 0;
    ip -> saddr = LocalHostIP;
    ip -> daddr = inet_addr(&HostIP[0]);

    //填充icmp头
    icmp = (struct icmphdr*)(ip + 1);
    icmp->type = ICMP_ECHO;
    icmp->code = 0;
    icmp->un.echo.id = htons(LocalPort);
    icmp->un.echo.sequence = 0;

    struct timeval *tp = (struct timeval*) &SendBuf[28];
    gettimeofday(tp, NULL);
    icmp -> checksum = in_cksum((u_short *)icmp, sizeof(struct icmphdr) +
sizeof(struct timeval));
```

```
    //设置套接字的发送地址
    struct sockaddr_in PingHostAddr;
    PingHostAddr.sin_family = AF_INET;
    PingHostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
    int Addrlen = sizeof(struct sockaddr_in);

    //发送ICMP请求
    ret = sendto(PingSock, SendBuf, SendBufSize, 0, (struct sockaddr*) &PingHostAddr,
sizeof(PingHostAddr));
    if(ret < 0) {
        std::cout << "sendto error" << std::endl;
        return false;
    }

    if(fcntl(PingSock, F_SETFL, O_NONBLOCK) == -1) {
        perror("fcntl error");
        return false;
    }

    struct timeval TpStart, TpEnd;
    bool flags;
    //循环等待接收ICMP响应
    gettimeofday(&TpStart, NULL); //获得循环起始时刻
    flags = false;

    char RecvBuf[1024];
    struct sockaddr_in FromAddr;
    struct icmp* Recvicmp;
    struct ip* Recvip;
    std::string SrcIP, DstIP, LocalIP;
    struct in_addr in_LocalhostIP;

    do {
        //接收ICMP响应
        ret = recvfrom(PingSock, RecvBuf, 1024, 0, (struct sockaddr*) &FromAddr,
        (socklen_t*) &Addrlen);
        if (ret > 0) //如果接收到一个数据包，对其进行解析
        {
            Recvip = (struct ip*) RecvBuf;
            Recvicmp = (struct icmp*) (RecvBuf + (Recvip -> ip_hl * 4));
            SrcIP = inet_ntoa(Recvip -> ip_src); //获得响应数据包IP头的源地址
            DstIP = inet_ntoa(Recvip -> ip_dst); //获得响应数据包IP头的目的地址
            in_LocalhostIP.s_addr = LocalHostIP;
            LocalIP = inet_ntoa(in_LocalhostIP); //获得本机IP地址
            //判断该数据包的源地址是否等于被测主机的IP地址，目的地址是否等于
            //本机IP地址，ICMP头的type字段是否为ICMP_ECHOREPLY
            if (SrcIP == HostIP && DstIP == LocalIP &&
            Recvicmp->icmp_type == ICMP_ECHOREPLY) {
                /*ping成功，退出循环*/
                std::cout << "Ping Host " << HostIP << " Successfully !" << std::endl;
                flags =true;
```

```
                break;
            }
        }
        //获得当前时刻，判断等待相应时间是否超过3秒，若是，则退出等待。
        gettimeofday(&TpEnd, NULL);
        float TimeUse = (1000000 * (TpEnd.tv_sec - TpStart.tv_sec) + (TpEnd.tv_usec -
TpStart.tv_usec)) / 1000000.0;
        if(TimeUse < 3) {
            continue;
        }
        else {
            flags = false;
            break;
        }
    } while(true);
    return flags;
}
```

TCP connect 扫描

我们这一部分的主要功能是利用 TCP 扫描确定目的主机的某一 TCP 端口是否开启，具体来收就是尝试连接被测主机的指定端口，若连接成功，则表示端口开启；否则，表示端口关闭。为了提高效率，我们使用多线程进行扫描，每个进程扫描一个端口。首先我们先创建两个线程锁，为了让我们扫描端口并行化扫描。使用变量 TCPConThrdNum 来记录已经创建的子线程数。然后我们编写 void* Thread_TCPconnectHost(void* param) 函数，该该函数的主要功能是连接目标主机指定端口的工作。首先，我们获得目标主机的 IP 地址和扫描端口号，然后创建流套接字，进入连接区，加锁防止多个线程同时打印字符出现乱码。然后设置连接主机，利用 connect 函数连接目标主机，加锁防止多个线程同时打印出现输出乱码，若连接成功，则表示端口开启；否则，表示端口关闭。

```
void* Thread_TCPconnectHost(void* param) {
    /*变量定义*/
    //获得目标主机的IP地址和扫描端口号
    struct TCPConHostThrParam *p = (struct TCPConHostThrParam*) param;
    std::string HostIP = p -> HostIP;
    unsigned HostPort = p -> HostPort;
    //创建流套接字
    int ConSock = socket(AF_INET,SOCK_STREAM,0);
    if(ConSock < 0) {
        pthread_mutex_lock(&TCPConPrintlocker);

    }

    //设置连接主机地址
    struct sockaddr_in HostAddr;
    memset(&HostAddr, 0, sizeof(HostAddr));
    HostAddr.sin_family = AF_INET;
    HostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
    HostAddr.sin_port = htons(HostPort);
    //connect目标主机
    int ret = connect(ConSock, (struct sockaddr*) &HostAddr, sizeof(HostAddr));
    if(ret < 0) {
        pthread_mutex_lock(&TCPConPrintlocker);
        std::cout << "TCP connect scan: " << HostIP << ":" << HostPort << " is closed"
<< std::endl;
        pthread_mutex_unlock(&TCPConPrintlocker);
    } else {
        pthread_mutex_lock(&TCPConPrintlocker);
        std::cout << "TCP connect scan: " << HostIP << ":" << HostPort << " is open"
<< std::endl;
        pthread_mutex_unlock(&TCPConPrintlocker);
    }
    delete p;
    close(ConSock); //关闭套接字
    //子线程数减1
    pthread_mutex_lock(&TCPConScanlocker);
    TCPConThrdNum--;
    pthread_mutex_unlock(&TCPConScanlocker);
} // TCP connect 扫描
```

编写 void* Thread_TCPconnectHost(void* param) 函数，该函数用于遍历目标主机的端口，是该功能的主线程函数。首先我们获得扫描的目标主机 IP、启始端口、终止端口，然后我们将线程数设置为 0。接下来，我们开始从起始端口到终止端口循环扫描目标主机的端口。首先我们在循环中设置子线程参数，然后将子线程设为分离状态，创建 connect 目标主机指定的端口和一个独立的子线程进行绑定，并将子线程数加 1。每一寸循环都会判断子线程的数量，如果如果子线程数大于 100，则暂时休眠。

```
void* Thread_TCPconnectScan(void* param)
{
    /*变量定义*/
    //获得扫描的目标主机IP，启始端口，终止端口
    struct TCPConThrParam *p = (struct TCPConThrParam*) param;
    std::string HostIP = p -> HostIP;
    unsigned BeginPort = p -> BeginPort;
    unsigned EndPort = p->EndPort;
    TCPConThrdNum = 0; //将线程数设为0
    //开始从起始端口到终止端口循环扫描目标主机的端口
    pthread_t subThreadID;
    pthread_attr_t attr;
    for (unsigned TempPort = BeginPort; TempPort <= EndPort; TempPort++)
    {
        //设置子线程参数
        TCPConHostThrParam *pConHostParam = new TCPConHostThrParam;
        pConHostParam->HostIP = HostIP;
        pConHostParam->HostPort = TempPort;
        //将子线程设为分离状态
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
        //创建connect目标主机指定的端口子线程
        int ret = pthread_create(&subThreadID, &attr, Thread_TCPconnectHost,
pConHostParam);
        if(ret == -1) {
            std::cout << "Create TCP connect scan thread error!" << std::endl;
        }
        //线程数加1
        pthread_mutex_lock(&TCPConScanlocker);
        TCPConThrdNum++;
        pthread_mutex_unlock(&TCPConScanlocker);
        //如果子线程数大于100，暂时休眠
        while (TCPConThrdNum>100) {
            sleep(3);
        }
    }
    //等待子线程数为0，返回
    while (TCPConThrdNum != 0) {
        sleep(1);
    }
    pthread_exit(NULL);
}
```

TCP SYN 扫描

我们这一部分的主要功能是利用 TCP SYN 扫描确定目的主机的某一 TCP 端口是否开启该，具体就是尝试向被测主机的指定端口发送 SYN 报文，如果接收到 ACK 报文，则说明开启，否则则说明关闭。为了提高效率，我们使用多线程进行扫描，每个进程扫描一个端口。首先我们先创建两个

线程锁，为了让我们扫描端口并行化扫描。使用变量 TCPSynThrdNum 来记录已经创建的子线程数。然后我们编写 void* Thread_TCPSYNHost(void* param) 函数，该函数的主要功能是完成对目标主机指定端口的 TCP SYN 扫描。首先，我们获得目标主机的 IP 地址和扫描端口号。

创建套接字，填充 TCP SYN 数据包，填充 TCP 伪头部，用于计算校验和，填充 TCP 头，封装 IP 头，发送 TCP SYN 数据包，开始利用一个循环循环接受包到 buffer 中。解析 IP 头和 TCP 头，然后从 TCP 头和 IP 头中解析源地址、目的地址、源 IP、目的 IP。判断响应数据包的源地址是否等于目标主机地址，目的地址是否等于本机；IP 地址，源端口是否等于被扫描端口，目的端口是否等于本机端口号。判断数据包类型，给出动作响应。只让这个过程循环 20 次，如果没收到默认关闭。

```
pthread_mutex_t TCPSynPrintlocker = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t TCPSynScanlocker = PTHREAD_MUTEX_INITIALIZER;

int TCPSynThrdNum;

void* Thread_TCPSYNHost(void* param) {
    /*变量定义*/
    //获得目标主机的IP地址和扫描端口号
    struct TCPSYNHostThrParam *p = (struct TCPSYNHostThrParam*) param;
    std::string HostIP = p -> HostIP;
    unsigned HostPort = p -> HostPort;
    unsigned LocalPort = p -> LocalPort;
    unsigned LocalHostIP = p -> LocalHostIP;

    struct sockaddr_in SYNScanHostAddr;
    memset(&SYNScanHostAddr, 0, sizeof(SYNScanHostAddr));
    SYNScanHostAddr.sin_family = AF_INET;
    SYNScanHostAddr.sin_addr.s_addr = inet_addr(HostIP.c_str());
    SYNScanHostAddr.sin_port = htons(HostPort);
    //创建套接字
    int SynSock = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
    if(SynSock < 0) {
        pthread_mutex_lock(&TCPSynPrintlocker);
        std::cout << "Can't creat raw socket !" << std::endl;
        pthread_mutex_unlock(&TCPSynPrintlocker);
    }
    int flag = 1;
    if (setsockopt(SynSock, IPPROTO_IP, IP_HDRINCL, (void*)&flag, sizeof(int)) ==
        -1) {
        std::cout << "set IP_HDRINCL error.\n";
    }
    //填充TCP SYN数据包
    char sendbuf[8192];
    char recvbuf[8192];
    struct pseudohdr *ptcph = (struct pseudohdr*) sendbuf;
struct tcphdr *tcph = (struct tcphdr*)(sendbuf + sizeof(struct pseudohdr));
    //填充TCP伪头部，用于计算校验和
    ptcph -> saddr = LocalHostIP;
    ptcph -> daddr = inet_addr(HostIP.c_str());
    in_addr src, dst;
    ptcph -> useless = 0;
    ptcph -> protocol = IPPROTO_TCP;
    ptcph -> length = htons(sizeof(struct tcphdr));

    src.s_addr = ptcph -> saddr;
    dst.s_addr = ptcph -> daddr;

    //填充TCP头
    memset(tcph, 0, sizeof(struct tcphdr));
    // std::cout<<LocalPort<<" "<<HostPort<<std::endl;
```

```
    tcph->th_sport = htons(LocalPort);
    tcph->th_dport = htons(HostPort);
    tcph->th_seq = htonl(123456);
    tcph->th_ack = 0;
    tcph->th_x2 = 0;
    tcph->th_off = 5;
    tcph->th_flags = TH_SYN;
    tcph->th_win = htons(65535);
    tcph->th_sum = 0;
    tcph->th_urp = 0;
    tcph->th_sum = in_cksum((unsigned short*)ptcph, 20 + 12);

    IPHeader IPheader(ptcph -> saddr, ptcph -> daddr, IPPROTO_TCP);
    char temp[sizeof(IPHeader) + sizeof(struct tcphdr)];

    memcpy((void*)temp, (void*)&IPheader, sizeof(IPheader));
memcpy((void*)(temp+sizeof(IPheader)), (void*)tcph, sizeof(struct tcphdr));

    //发送TCP SYN数据包
    int len = sendto(SynSock, temp, sizeof(IPHeader) + sizeof(struct tcphdr), 0,
(struct sockaddr *)&SYNScanHostAddr, sizeof(SYNScanHostAddr));
    // std::cout << sizeof(IPHeader) <<" "<< sizeof(struct tcphdr)<<" "<<len <<
std::endl;
    if(len < 0) {
        pthread_mutex_lock(&TCPSynPrintlocker);
        std::cout << "Send TCP SYN Packet error !" << std::endl;
        pthread_mutex_unlock(&TCPSynPrintlocker);
    }
    int count = 0;
    std::string SrcIP;
    struct ip *iph;
    flag = 1;
    sockaddr_in recvAddr;
    int addrLen = sizeof(recvAddr);
    do{
        len = recvfrom(SynSock, recvbuf, 8192, 0, (sockaddr*)&recvAddr,
                       (socklen_t*)&addrLen);
        if(len < 0) {
            /*接收错误*/
            pthread_mutex_lock(&TCPSynPrintlocker);
            std::cout << "Read TCP SYN Packet error !" << std::endl;
            pthread_mutex_unlock(&TCPSynPrintlocker);
        }
        else {
            struct ip *iph = (struct ip *)recvbuf;
            int i = iph -> ip_hl * 4;
            tcph = (struct tcphdr *)(recvbuf + i);

            std::string SrcIP = inet_ntoa(iph -> ip_src);
            std::string DstIP = inet_ntoa(iph -> ip_dst);
            struct in_addr in_LocalhostIP;
```

```
            in_LocalhostIP.s_addr = LocalHostIP;
            std::string LocalIP = inet_ntoa(in_LocalhostIP);

            unsigned SrcPort = ntohs(tcph -> th_sport);
            unsigned DstPort = ntohs(tcph -> th_dport);
            if(HostIP == SrcIP && LocalIP == DstIP && SrcPort == HostPort && DstPort
== LocalPort)
            {
                // std::cout<<(int)(tcph->th_flags)<<std::endl;
                if(tcph->th_flags == 0x12) //判断是否为SYN|ACK数据包
                {           /*端口开启*/ flag = 0;
                    pthread_mutex_lock(&TCPSynPrintlocker);
std::cout << "Host: " << SrcIP << " Port: " << ntohs(tcph -> th_sport)
<< " open !" << std::endl;     pthread_mutex_unlock(&TCPSynPrintlocker);
                }
                if(tcph->th_flags == 0x14) //判断是否为RST数据包
                {
                    /*端口关闭*/
                    flag = 0;
                    pthread_mutex_lock(&TCPSynPrintlocker);
                    std::cout << " Port: " << ntohs(tcph -> th_sport) << " closed !"
<< std::endl;
                    pthread_mutex_unlock(&TCPSynPrintlocker);
                }
            }
        }
    } while(count++ < 20 && flag);
    //退出子线程
    if(flag){
        pthread_mutex_lock(&TCPSynPrintlocker);  std::cout << "Host: " <<
 SrcIP << " Port: " << HostPort << " closed !" << std::endl;
        pthread_mutex_unlock(&TCPSynPrintlocker);
    }
    delete p;
    close(SynSock);
    pthread_mutex_lock(&TCPSynScanlocker);
    TCPSynThrdNum--;
    pthread_mutex_unlock(&TCPSynScanlocker);
}
```

然后我们进行编写 void* Thread_TCPSynScan(void* param) 函数，该函数的主要功能是调用 Thread_TCPSYNHost 函数创建多个扫描子线程负责遍历目标主机的被测端口。首先我们获得目标主机的 IP 地址和扫描的起始端口号，终止端口号，以及本机的 IP 地址。接下来，我们开始从起始端口到终止端口循环扫描目标主机的端口。首先我们在循环中设置子线程参数，然后将子线程设为分离状态，创建 SYN 目标主机指定的端口和一个独立的子线程进行绑定，并将子线程数加 1。每一寸循环都会判断子线程的数量, 如果如果子线程数大于 100，则暂时休眠。

```
void* Thread_TCPSynScan(void* param) {
    struct TCPSYNThrParam *p = (struct TCPSYNThrParam*)param;
    std::string HostIP = p -> HostIP;
    unsigned BeginPort = p-> BeginPort;
    unsigned EndPort = p-> EndPort;
    unsigned LocalHostIP = p -> LocalHostIP;
    //循环遍历扫描端口
    TCPSynThrdNum = 0;
    unsigned LocalPort = 1024;
    pthread_attr_t attr,lattr;
    pthread_t listenThreadID,subThreadID;
    for (unsigned TempPort = BeginPort; TempPort <= EndPort; TempPort++)
    {
        //设置子线程参数
        struct TCPSYNHostThrParam *pTCPSYNHostParam =
        new TCPSYNHostThrParam;
        pTCPSYNHostParam->HostIP = HostIP;
        pTCPSYNHostParam->HostPort = TempPort;
        pTCPSYNHostParam->LocalPort = TempPort + LocalPort;
        pTCPSYNHostParam->LocalHostIP = LocalHostIP;
        //将子线程设置为分离状态
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
        //创建子线程
        int ret = pthread_create(&subThreadID, &attr, Thread_TCPSYNHost,
pTCPSYNHostParam);
        if (ret==-1)
        {
            std::cout << "Can't create the TCP SYN Scan Host thread !" << std::endl;
        }
        pthread_attr_destroy(&attr);
        //子线程数加1
        pthread_mutex_lock(&TCPSynScanlocker);
        TCPSynThrdNum++;
        pthread_mutex_unlock(&TCPSynScanlocker);
        //子线程数大于100，休眠
        while(TCPSynThrdNum > 100) {
            sleep(3);
        }
    }
    while(TCPSynThrdNum != 0) {
        sleep(1);
    }
    //返回主流程
    pthread_exit(NULL);
}
```

TCP FIN 扫描

我们这一部分的主要功能是利用 TCP FIN 扫描确定目的主机的某一 TCP 端口是否开启，具体来说就尝试向被测主机发送 FIN 报文，如果接收到 ACK 报文，则说明开启，否则则说明关闭。未来提高效率，我们使用多线程进行扫描，每个进程扫描一个端口。首先我们先创建两个线程锁，为了让我们扫描端口并行化进行。使用变量 TCPFinThrdNum 来记录已经创建的子进程数。然后我们编写 void* Thread_TCPFINHost(void* param) 函数，该函数的主要目的是完成对目标主机制定端口的 TCP FIN 扫描。首先，我们获得目标主机的 IP 地址和扫描端口号。然后我们创建两个套接字，一个用于发送 FIN 报文，一个用于接收回复。

填充 TCP 头,封装 IP 头,发送 TCP FIN 数据包，并将另一个套接字设置为非阻塞模式进行接收,然后开始利用一个循环将收到的数据包存入 buffer 中,然后我们需要判断响应包的原地址是不是等于目的主机地址，并解析 IP 头和 TCP 头，然后从TCP 头和 IP 头中解析源地址、目的地址、源 IP、目的 IP。判断响应数据包的源地址是否等于目标主机，目的地址是否等于本地主机；IP 地址，源端口是否等于被扫描端口，目的端口是否等于本机号。然后我们判断数据包类型，给出动作响应，只让这个过程重复 5 秒，如果 5 秒内没有响应则判定为没有响应，继续进行扫描。

```
void* Thread_TCPFINHost(void* param) {
    /*------------与 TCP SYN 扫描类似-----------------*/
    //填充 TCP FIN 数据包
    struct TCPFINHostThrParam *p = (struct TCPFINHostThrParam*)param;
    std::string HostIP = p -> HostIP;
    unsigned HostPort = p->HostPort;
    unsigned LocalPort = p->LocalPort;
    unsigned LocalHostIP = p->LocalHostIP;
    struct sockaddr_in FINScanHostAddr;
    memset(&FINScanHostAddr, 0, sizeof(FINScanHostAddr));
    FINScanHostAddr.sin_family = AF_INET;
    FINScanHostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
    FINScanHostAddr.sin_port = htons(HostPort);

    int FinSock=socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
    if (FinSock < 0) {
        pthread_mutex_lock(&TCPFinPrintlocker);
        std::cout << "Can't creat raw socket !" << std::endl;
        pthread_mutex_unlock(&TCPFinPrintlocker);
    }

    int FinRevSock = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
    if (FinRevSock < 0)
    {
        pthread_mutex_lock(&TCPFinPrintlocker);
        std::cout << "Can't creat raw socket !" << std::endl;
        pthread_mutex_unlock(&TCPFinPrintlocker);
    }
    int flag = 1;    if (setsockopt(FinSock, IPPROTO_IP, IP_HDRINCL,
(void*)&flag, sizeof(int)) ==  -1) {
        std::cout << "set IP_HDRINCL error.\n";    }
    if (setsockopt(FinRevSock, IPPROTO_IP, IP_HDRINCL, (void*)&flag,
sizeof(int)) ==    -1) {
        std::cout << "set IP_HDRINCL error.\n";
    }
    char sendbuf[8192];
    struct pseudohdr *ptcph = (struct pseudohdr*)sendbuf;
    struct tcphdr *tcph = (struct tcphdr*)(sendbuf + sizeof(struct
pseudohdr));
    ptcph->saddr = LocalHostIP;
    ptcph->daddr = inet_addr(&HostIP[0]);
    ptcph->useless = 0;
    ptcph->protocol = IPPROTO_TCP;
    ptcph->length = htons(sizeof(struct tcphdr));
    tcph->th_sport = htons(LocalPort);
    tcph->th_dport = htons(HostPort);
    tcph->th_seq = htonl(123456);
    tcph->th_ack = 0;
    tcph->th_x2 = 0;
    tcph->th_off = 5;
```

```
    tcph->th_flags = TH_FIN;
    tcph->th_win = htons(65535);
    tcph->th_sum = 0;
    tcph->th_urp = 0;
    tcph->th_sum = in_cksum((unsigned short*)ptcph, 20 + 12);
    IPHeader IPheader(ptcph -> saddr, ptcph -> daddr, IPPROTO_TCP);
    char temp[sizeof(IPHeader) + sizeof(struct tcphdr)];
    memcpy((void*)temp, (void*)&IPheader, sizeof(IPheader));
memcpy((void*)(temp+sizeof(IPheader)), (void*)tcph, sizeof(struct tcphdr));
    //发送 TCP FIN 数据包
int len = sendto(FinSock, temp, sizeof(IPHeader) + sizeof(struct tcphdr), 0, (struct
sockaddr *)&FINScanHostAddr, sizeof(FINScanHostAddr));
    if(len < 0)
    {
        pthread_mutex_lock(&TCPFinPrintlocker);
        std::cout << "Send TCP FIN Packet error !" << std::endl;
        pthread_mutex_unlock(&TCPFinPrintlocker);
    }
    //将套接字设置为非阻塞模式
    if(fcntl(FinRevSock, F_SETFL, O_NONBLOCK) == -1)
    {
        pthread_mutex_lock(&TCPFinPrintlocker);
        std::cout << "Set socket in non-blocked model fail !" << std::endl;
        pthread_mutex_unlock(&TCPFinPrintlocker);
    }
    int FromAddrLen = sizeof(struct sockaddr_in);
    //接收 TCP 响应数据包循环
    struct timeval TpStart,TpEnd;
    char recvbuf[8192];
    struct sockaddr_in FromAddr;
    std::string SrcIP, DstIP, LocalIP;
    gettimeofday(&TpStart, NULL); //获得开始接收时刻
    struct in_addr in_LocalhostIP;
    do {
        //调用 recvfrom 函数接收数据包
        len = recvfrom(FinRevSock, recvbuf, sizeof(recvbuf), 0,
(struct sockaddr*) &FromAddr, (socklen_t*) &FromAddrLen);
        if(len > 0)        {
            std::string SrcIP = inet_ntoa(FromAddr.sin_addr);
            if(1)
            {                       //响应数据包的源地址等于目标主机地址
                struct ip *iph = (struct ip *)recvbuf;
                int i = iph -> ip_hl * 4;
                struct tcphdr *tcph = (struct tcphdr *)&recvbuf[i];
                SrcIP = inet_ntoa(iph->ip_src);
                DstIP = inet_ntoa(iph->ip_dst);
                in_LocalhostIP.s_addr = LocalHostIP;
                LocalIP = inet_ntoa(in_LocalhostIP);
                unsigned SrcPort = ntohs(tcph->th_sport);
                unsigned DstPort = ntohs(tcph->th_dport);
                //判断响应数据包的源地址是否等于目标主机地址，目的地址是否等于本机 IP 地
```

```
址，源端口是否等于被扫描端口，目的端口是否等于本机端口号
if(HostIP == SrcIP && LocalIP == DstIP && SrcPort == HostPort &&
DstPort == LocalPort)
                {                    //判断是否为 RST 数据包
                    if (tcph->th_flags == 0x14)
                    {      pthread_mutex_lock(&TCPFinPrintlocker);
std::cout << "Host: " << SrcIP << " Port: " << ntohs(tcph -> th_sport)
<< " closed !" << std::endl;pthread_mutex_unlock(&TCPFinPrintlocker);}
                    break;    }         }        }
        //判断等待响应数据包时间是否超过 3 秒
gettimeofday(&TpEnd, NULL);float TimeUse = (1000000 * (TpEnd.tv_sec
 - TpStart.tv_sec) + (TpEnd.tv_usec - TpStart.tv_usec)) / 1000000.0;
        if(TimeUse < 5)
        {            continue;        }
        else        {
            //超时，扫描端口开启
            pthread_mutex_lock(&TCPFinPrintlocker);
            std::cout << "Host: " << HostIP << " Port: " << HostPort << " open !" <<
std::endl;
            pthread_mutex_unlock(&TCPFinPrintlocker);
            break;        }    }
    while(true);    //退出子线程
    delete p;    close(FinSock);
    close(FinRevSock);
    pthread_mutex_lock(&TCPFinScanlocker);
    TCPFinThrdNum--;
    pthread_mutex_unlock(&TCPFinScanlocker);
}
```

然后我们编写 void* Thread_TCPFinScan(void* param) 函数，该函数的主要功能是调用 Thread_TCPFINHost函数创建多个扫描子线程负责遍历目标主机的被测端口。首先我们获取目标主机的 IP 地址和扫描的起始端口号，终止端口号，以及本机的 IP地址。

接下来, 我们开始从起始端口到终止端口循环扫描目标主机的端口。首先我们在循环中设置子线程参数，然后将子线程设为分离状态。创建 FIN 目标主机指定的端口和一个独立的子线程进行绑定，并将子线程数加 1。每一次循环都会判断子线程的数量, 如果如果子线程数大于 100，则暂时休眠。

```
void* Thread_TCPFinScan(void* param) {
    struct TCPFINThrParam *p = (struct TCPFINThrParam*)param;
    std::string HostIP = p->HostIP;
    unsigned BeginPort = p->BeginPort;
    unsigned EndPort = p->EndPort;
    unsigned LocalHostIP = p->LocalHostIP;
    TCPFinThrdNum = 0;
    unsigned LocalPort = 1024;
    pthread_attr_t attr, lattr;
    pthread_t listenThreadID, subThreadID;
    for (unsigned TempPort = BeginPort; TempPort <= EndPort;TempPort++)
    {
struct TCPFINHostThrParam *pTCPFINHostParam = new TCPFINHostThrParam;
        pTCPFINHostParam->HostIP = HostIP;
        pTCPFINHostParam->HostPort = TempPort;
        pTCPFINHostParam->LocalPort = TempPort + LocalPort;
        pTCPFINHostParam->LocalHostIP = LocalHostIP;
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
        int ret = pthread_create(&subThreadID,&attr,
Thread_TCPFINHost,pTCPFINHostParam);
        if (ret == -1)
        {std::cout << "Can't create the TCP FIN Scan Host thread !"
<< std::endl;    }
        pthread_attr_destroy(&attr);
        pthread_mutex_lock(&TCPFinScanlocker);
        TCPFinThrdNum++;
        pthread_mutex_unlock(&TCPFinScanlocker);
        while (TCPFinThrdNum>100)
        {
            sleep(3);
        }
    }
    while (TCPFinThrdNum != 0)
    {
        sleep(1);
    }
    std::cout << "TCP FIN scan thread exit !" << std::endl;
    pthread_exit(NULL);
}
```

UDP 扫描

我们这一部分的主要功能是利用 UDP 扫描确定目的主机的 UDP 扫描确定目的主机的某一 UDP端口是否开启，具体来说就是向被测主机的指定端口发送 UDP 报文，如果收到回复则说明连接成果，则表示端口开启；否则，表示端口关闭。但是这个扫描和之前的有所不同，因为目标主机返回的 ICMP 不可达数据包没有包含目标主机的源端口号，扫描器无法判断 ICMP 响应是从哪个端

口发出的，如果让让多个子线程同时扫描端口，会造成无法区分 ICMP 响应数据包与其对应端口的情况，无法进行判断。所以我们舍弃了多线程并行的操作，可能效率有所降低，但是准确率有所保障。我们首先编写 void* UDPScanHost(void* param) 函数，该函数的主要功能是完成主机对指定端口的扫描。在该函数最开始，首先获得目标主机的 IP 地址和扫描端口号，然后创建流套接字，发送信息。

然后设置 UDP 套接字地址，填充 UDP 数据包，填充 UDP 头和伪首部，用于计算校验和。填充 IP 头，然后发送 UDP 数据包，并将 UDPSock 套接字设置为为非阻塞模式，开始利用一个循环循环接受包到 buffer 中，如果接收到了则说明端口打开，如果没接收到则说明没有打开。判断接收时间，如果接收时间超过了三秒没有响应则自动认为没有链接，这时候便退出。

```
void* UDPScanHost(void* param) {
    struct UDPScanHostThrParam *p = (struct UDPScanHostThrParam*) param;
    std::string HostIP = p -> HostIP;
    unsigned HostPort = p -> HostPort;
    unsigned LocalPort = p -> LocalPort;
    unsigned LocalHostIP = p -> LocalHostIP;
    int UDPSock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if(UDPSock < 0)    {
        pthread_mutex_lock(&UDPPrintlocker);
        std::cout << "Can't creat raw icmp socket !" << std::endl;
        pthread_mutex_unlock(&UDPPrintlocker);    }
    int on = 1;
int ret = setsockopt(UDPSock, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));

    if (ret < 0)
    {        pthread_mutex_lock(&UDPPrintlocker);
        std::cout << "Can't set raw socket !" << std::endl;
        pthread_mutex_unlock(&UDPPrintlocker);    }
    struct sockaddr_in UDPScanHostAddr;
    memset(&UDPScanHostAddr, 0, sizeof(UDPScanHostAddr));
    UDPScanHostAddr.sin_family = AF_INET;
    UDPScanHostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
    UDPScanHostAddr.sin_port = htons(HostPort);
    char packet[sizeof(struct iphdr) + sizeof(struct udphdr)];
    memset(packet, 0x00, sizeof(packet));
    struct iphdr *ip = (struct iphdr *)packet;
    struct udphdr *udp = (struct udphdr *)(packet + sizeof(struct iphdr));
struct pseudohdr *pseudo = (struct pseudohdr *)(packet + sizeof(struct iphdr) -
sizeof(struct pseudohdr));
    udp -> source = htons(LocalPort);
    udp -> dest = htons(HostPort);
    udp -> len = htons(sizeof(struct udphdr));
    udp -> check = 0;
    pseudo -> saddr = LocalHostIP;
    pseudo -> daddr = inet_addr(&HostIP[0]);
    pseudo -> useless = 0;
    pseudo -> protocol = IPPROTO_UDP;
    pseudo -> length = udp->len;
udp->check = in_cksum((u_short *)pseudo,sizeof(struct udphdr)+
sizeof(struct pseudohdr));
    ip -> ihl = 5;     ip -> version = 4;
    ip -> tos = 0x10;     ip -> tot_len = sizeof(packet);
    ip -> frag_off = 0;     ip -> ttl = 69;
    ip -> protocol = IPPROTO_UDP;     ip -> check = 0;
    ip -> saddr = inet_addr("192.168.1.168");
    ip -> daddr = inet_addr(&HostIP[0]);
    int n = sendto(UDPSock, packet, ip -> tot_len, 0,
(struct sockaddr *)&UDPScanHostAddr, sizeof(UDPScanHostAddr));
    if (n < 0)     {
        pthread_mutex_lock(&UDPPrintlocker);
```

```
        std::cout << "Send message to Host Failed !" << std::endl;
        pthread_mutex_unlock(&UDPPrintlocker);    }
    if(fcntl(UDPSock, F_SETFL, O_NONBLOCK) == -1)
    {       pthread_mutex_lock(&UDPPrintlocker);
std::cout << "Set socket in non-blocked model fail !" << std::endl;
        pthread_mutex_unlock(&UDPPrintlocker);    }
    struct timeval TpStart, TpEnd;
    struct ipicmphdr hdr;
    gettimeofday(&TpStart,NULL);            //get start time
    do     {        //receive response message
        n = read(UDPSock, (struct ipicmphdr *)&hdr, sizeof(hdr));
        if(n > 0)
        {    if((hdr.ip.saddr == inet_addr(&HostIP[0])) &&
(hdr.icmp.code == 3) && (hdr.icmp.type == 3))
            {    pthread_mutex_lock(&UDPPrintlocker);std::cout << "Host: "
 << HostIP << " Port: " << HostPort << " closed !" << std::endl;
                pthread_mutex_unlock(&UDPPrintlocker);        break;
            }        }
        //time out?
gettimeofday(&TpEnd,NULL);float TimeUse = (1000000 * (TpEnd.tv_sec -
TpStart.tv_sec) + (TpEnd.tv_usec - TpStart.tv_usec)) / 1000000.0;
        if(TimeUse < 3)        {
            continue;        }
        else
        {            pthread_mutex_lock(&UDPPrintlocker);
            std::cout << "Host: " << HostIP << " Port: " << HostPort
<< " open !" << std::endl;
            pthread_mutex_unlock(&UDPPrintlocker);        break;
        }    }
    while(true);
    //close socket
    close(UDPSock);
    delete p;
}
```

然后我们编写 void* Thread_UDPScan(void* param) 函数，该函数的作用是调用 UDPScanHost函数创建线程负责遍历目标主机被测端口，首先我们获取目标主机的 IP 地址和扫描的起始端口号，终止端口号，以及本机的 IP 地址。

接下来，我们从起始端口到终止端口循环遍历目标主机的端口，进行对端口的扫描。

```
void* Thread_UDPScan(void* param)
{struct UDPThrParam *p = (struct UDPThrParam*) param;
    std::string HostIP = p -> HostIP;
    unsigned BeginPort = p -> BeginPort;
    unsigned EndPort = p -> EndPort;
    unsigned LocalHostIP = p -> LocalHostIP;
    unsigned LocalPort = 1024;
    for (unsigned TempPort = BeginPort; TempPort <= EndPort; TempPort++)
    {
        UDPScanHostThrParam *pUDPScanHostParam = new UDPScanHostThrParam;
        pUDPScanHostParam->HostIP = HostIP;
        pUDPScanHostParam->HostPort = TempPort;
        pUDPScanHostParam->LocalPort = TempPort + LocalPort;
        pUDPScanHostParam->LocalHostIP = LocalHostIP;
        UDPScanHost(pUDPScanHostParam);    }
    std::cout << "UDP Scan thread exit !" << std::endl;
    pthread_exit(NULL);}
```

端口扫描器程序 Scaner

首先程序判断是否需要输出帮助信息，若是，则输出端口扫描器程序的帮助信息，然后退出；否则，执行下面的步骤。

```
int main(int argc,char *argv[]) { std::unordered_map<std::string, void(*)
(int, char*[])> mapOp = {{"-h", print_h}, {"-c", print_c}, {"-s", print_s}
, {"-u", print_u}, {"-f", print_f}};    if (argc != 2) {
std::cout << "参数错误，argc = " << argc << std::endl;    return -1;    }
    std::string op = argv[1];    if(op != "-h") {
        std::cout << "Please input IP address of a Host:";
        std::cin >> HostIP;
        if(inet_addr(&(HostIP[0])) == INADDR_NONE)        {
            std::cout << "IP address wrong!" << std::endl;
            return -1;        }
        std::cout << "Please input the range of port..." << std::endl;
        std::cout << "Begin Port:";
        std::cin >> BeginPort;
        std::cout << "End Port:";
        std::cin >> EndPort;
        if(BeginPort > EndPort) {
            std::cout << "The range of port is wrong !" << std::endl;
            return -1;        }
        else        {if(BeginPort < 1 || BeginPort > 65535 || EndPort
< 1 || EndPort > 65535) {std::cout << "The range of port is wrong !"
<< std::endl;    return -1;        }
            else {std::cout << "Scan Host " << HostIP << " port "
 << BeginPort << "~" << EndPort << " ..." << std::endl;  }   }
        if(!Ping(HostIP, GetLocalHostIP())) {
     std::cout << "Ping Host " << HostIP << " Failed !" << std::endl;
            return -1;      }
    }
    if (mapOp.find(op) != mapOp.end()) {
        mapOp[op](argc, argv);
        return 0;
    }
    return 0;
}
```

输入目的 IP 地址和端口信息，并判断是否合法。并依次执行选择下列函数。

TCP connect 扫描函数

```
void print_c(int argc, char *argv[]) {
    std::cout << "Begin TCP connect scan..." << std::endl;
    // struct TCPConThrParam TCPConParam;
    TCPConParam.HostIP = HostIP;
    TCPConParam.BeginPort = BeginPort;
    TCPConParam.EndPort = EndPort;int ret = pthread_create(&ThreadID,
NULL,Thread_TCPconnectScan, &TCPConParam);    if (ret==-1) {
        std::cout << "Can't create the TCP connect scan thread !" << std::endl;
        return;    }
    ret = pthread_join(ThreadID, NULL);
    if(ret != 0) {
        std::cout << "call pthread_join function failed !" << std::endl;
        return;    }
    else { std::cout << "TCP Connect Scan finished !" << std::endl;   }}
```

TCP SYN 函数

```
void print_s(int arg, char *argv[]) {
    std::cout << "Begin TCP SYN scan..." << std::endl;
    //create thread for TCP SYN scan
    // struct TCPSYNThrParam TCPSynParam;
    TCPSynParam.HostIP = HostIP;
    TCPSynParam.BeginPort = BeginPort;
    TCPSynParam.EndPort = EndPort;
    TCPSynParam.LocalHostIP = GetLocalHostIP();int ret =
pthread_create(&ThreadID, NULL, Thread_TCPSynScan, &TCPSynParam);
    if (ret == -1)     {
      std::cout << "Can't create the TCP SYN scan thread !" << std::endl;
        return;    }
    ret = pthread_join(ThreadID, NULL);
    if(ret != 0)    {
       std::cout << "call pthread_join function failed !" << std::endl;
        return;    }    else
    {      std::cout << "TCP SYN Scan finished !" << std::endl;
        return;    }}
```

TCP FIN 函数

```
void print_f(int argc, char *argv[]) {
    std::cout << "Begin TCP FIN scan..." << std::endl;
    //create thread for TCP FIN scan
    TCPFinParam.HostIP = HostIP;
    TCPFinParam.BeginPort = BeginPort;
    TCPFinParam.EndPort = EndPort;
    TCPFinParam.LocalHostIP = GetLocalHostIP();
    ret = pthread_create(&ThreadID, NULL, Thread_TCPFinScan, &TCPFinParam);
    if (ret==-1)
    {
        std::cout << "Can't create the TCP FIN scan thread !" << std::endl;
        return;
    }
    ret = pthread_join(ThreadID,NULL);
    if(ret != 0)
    {
        std::cout << "call pthread_join function failed !" << std::endl;
        return;
    }
    else
    {
        std::cout <<"TCP FIN Scan finished !" << std::endl;
        return;
    }
}
```

UDP 函数

```
void print_u(int arg, char *argv[]) {
    std::cout << "Begin UDP scan..." << std::endl;
    //create thread for UDP scan
    UDPParam.HostIP = HostIP;
    UDPParam.BeginPort = BeginPort;
    UDPParam.EndPort = EndPort;
    UDPParam.LocalHostIP = LocalHostIP;
    ret = pthread_create(&ThreadID, NULL, Thread_UDPScan, &UDPParam);
    if (ret == -1)
    {   std::cout << "Can't create the UDP scan thread !" << std::endl;
        return;    }
    ret = pthread_join(ThreadID,NULL);
    if(ret != 0)    {
        std::cout << "call pthread_join function failed !" << std::endl;
        return;    }    else    {
        std::cout << "UDP Scan finished !" << std::endl;
        return;   }}
```

四、实验结果

1、打印帮助函数

```
starx@starx-ubuntu:~/桌面/work4/code/bin$ sudo ./Scaner -h
[sudo] starx 的密码：
Scaner: usage:
        [-h] --help information
        [-c] --TCP connect scan
        [-s] --TCP syn scan
        [-f] --TCP fin scan
        [-u] --UDP scan
```

2、TCPconnect扫描

```
starx@starx-ubuntu:~/桌面/新建文件夹/code/bin$ sudo ./Scaner -c
[sudo] starx 的密码：
Please input IP address of a Host:127.0.0.1
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 127.0.0.1 port 1~255 ...
Ping Host 127.0.0.1 Successfully !
Begin TCP connect scan...
TCP connect scan: 127.0.0.1:1 is closed
TCP connect scan: 127.0.0.1:2 is closed
TCP connect scan: 127.0.0.1:5 is closed
TCP connect scan: 127.0.0.1:16 is closed
TCP connect scan: 127.0.0.1:9 is closed
TCP connect scan: 127.0.0.1:17 is closed
TCP connect scan: 127.0.0.1:6 is closed
TCP connect scan: 127.0.0.1:20 is closed
TCP connect scan: 127.0.0.1:7 is closed
TCP connect scan: 127.0.0.1:18 is closed
TCP connect scan: 127.0.0.1:8 is closed
TCP connect scan: 127.0.0.1:11 is closed
TCP connect scan: 127.0.0.1:10 is closed
TCP connect scan: 127.0.0.1:13 is closed
```

3、UDP扫描

```
starx@starx-ubuntu:~/桌面/新建文件夹/code/bin$ sudo ./Scaner -u
[sudo] starx 的密码:
Please input IP address of a Host:127.0.0.1
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 127.0.0.1 port 1~255 ...
Ping Host 127.0.0.1 Successfully !
Begin UDP scan...
Host: 127.0.0.1 Port: 1 open !
Host: 127.0.0.1 Port: 2 open !
Host: 127.0.0.1 Port: 3 open !
Host: 127.0.0.1 Port: 4 open !
Host: 127.0.0.1 Port: 5 open !
Host: 127.0.0.1 Port: 6 open !
Host: 127.0.0.1 Port: 7 open !
Host: 127.0.0.1 Port: 8 open !
Host: 127.0.0.1 Port: 9 open !
Host: 127.0.0.1 Port: 10 open !
Host: 127.0.0.1 Port: 11 open !
Host: 127.0.0.1 Port: 12 open !
Host: 127.0.0.1 Port: 13 open !
Host: 127.0.0.1 Port: 14 open !
Host: 127.0.0.1 Port: 15 open !
```

4.TCPSYN扫描


starx@starx-ubuntu:~/桌面/新建文件夹/code/bin$ sudo ./Scaner -s
[sudo] starx 的密码:
对不起，请重试。
[sudo] starx 的密码:
Please input IP address of a Host:127.0.0.1
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 127.0.0.1 port 1~255 ...
Ping Host 127.0.0.1 Successfully !
Begin TCP SYN scan...
Port: 3 closed !
Port: 2 closed !
Port: 1 closed !
Port: 8 closed !
Port: 7 closed !
Port: 9 closed !
Port: 6 closed !
Port: 13 closed !
Port: 15 closed !
Port: 4 closed !
Port: 11 closed !

5.TCPFIN扫描

```
starx@starx-ubuntu:~/桌面/新建文件夹/code/bin$ sudo ./Scaner -f
[sudo] starx 的密码:
Please input IP address of a Host:127.0.0.1
Please input the range of port...
Port:1
End Port:255
Scan Host 127.0.0.1 port 1~255 ...
Ping Host 127.0.0.1 Successfully !
Begin TCP FIN scan...
Host: 127.0.0.1 Port: 1 closed !
Host: 127.0.0.1 Port: 4 closed !
Host: 127.0.0.1 Port: 3 closed !
Host: 127.0.0.1 Port: 2 closed !
Host: 127.0.0.1 Port: 5 closed !
Host: 127.0.0.1 Port: 7 closed !
Host: 127.0.0.1 Port: 8 closed !
Host: 127.0.0.1 Port: 10 closed !
Host: 127.0.0.1 Port: 11 closed !
Host: 127.0.0.1 Port: 13 closed !
Host: 127.0.0.1 Port: 15 closed !
Host: 127.0.0.1 Port: 14 closed !
Host: 127.0.0.1 Port: 16 closed !
Host: 127.0.0.1 Port: 17 closed !
Host: 127.0.0.1 Port: 12 closed !
```

终端

五、出现的问题

权限问题：

最开始我们程序在创建套接字时创建失败，导致程序直接退出，后来意识到，是程序权限不够，于是在运行时加入 sudo 指令，问题解决。

数据格式问题：

最开始，我们的在进行 TCP 的 FIN 和 SYN 扫描的时候，无法进行扫描。然后我们进行打印，发现目的主机地址和源主机地址根本不匹配。查看我们的 IP 和 TCP 数据头数据结构，发现我错误的将uint32_t 类型的数据定义成了 int，实际上是转换成了 uint16_t 类型的数据，应该是 unsigned_int，这样就会导致我们的数据包的头部长度不正确，导致解析出来的东西是错误的。

总的来说就是因为系统的问题，有些 32 位的数据被定义为 64 位了，有些 16 位的数据被定义为 32 位。当我们更正了这个问题后，一切恢复正常。

六、实验总结

本次实验虽然给出了足够的框架代码满足我们实验的编写，但是一方面对linux的socket编程认知不足，一方面对windowssocket编程如何进行迁移还有一定的不熟悉，因此我们需要先进行了大量的关于其中编程的学习，最终我们成功编写到了相应的代码，完成了本次实验，感到受益匪浅。通过本次学习，我更深刻的领会到了网络通信以及其中蕴含的安全知识，感谢张老师和助教的殷勤付出，这门课程使我对之前所学的计算机网络和密码学课程巩固的同时，能够学习领会到更加全新的只是，使我受益匪浅。