

# 网络安全技术

## 实 验 报 告

学 院 网安学院  
年 级 21 级  
班 级 信息安全一班  
学 号 2113662  
姓 名 张丛  
手机号 18086215842

2024 年 5 月 28 日

# 目录

|                       |    |
|-----------------------|----|
| 一、实验目标 .....          | 1  |
| 二、实验内容 .....          | 1  |
| 三、实验步骤 .....          | 2  |
| 四、实验遇到的问题及其解决方法 ..... | 26 |
| 五、实验结论 .....          | 29 |

## 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。

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。

## 二、实验内容

在 Linux 环境下编写一个端口扫描器，利用套接字(socket)正确实现 ping 程序、TCP connect 扫描、TCP SYN 扫描、TCP FIN 扫描、以及 UDP 扫描。

ping 程序在用户输入被扫描主机 IP 地址之后探测该主机是否可达。

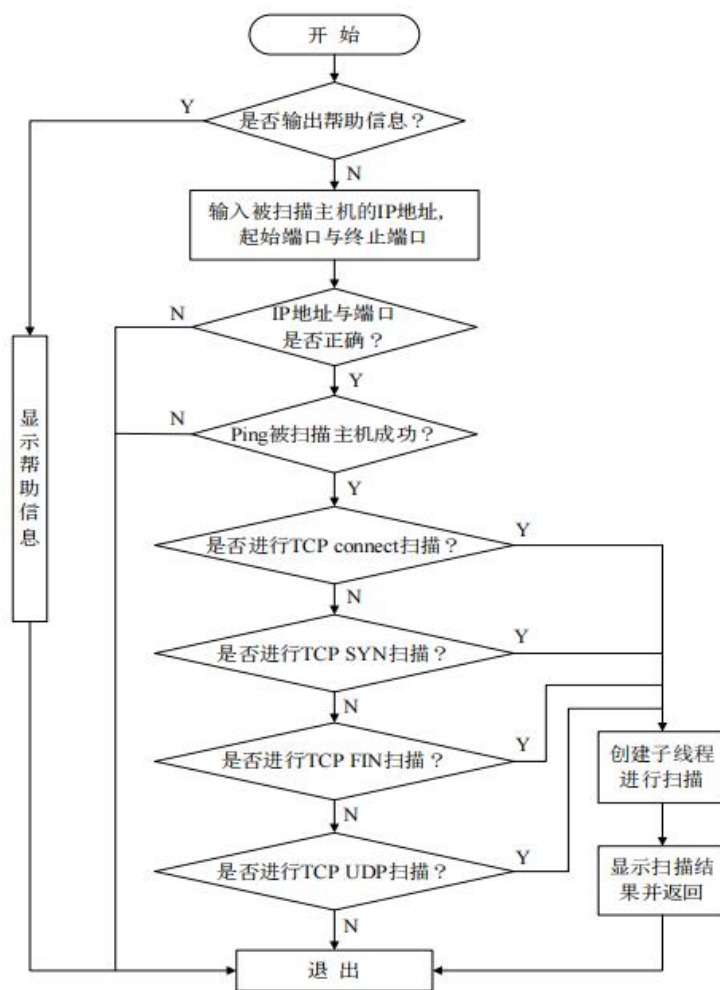
其它四种扫描在指定被扫描主机 IP，起始端口以及终止端口之后，从起始端口到终止端口对被测主机进行扫描。最后将每一个端口的扫描结果正确地显示出来。

### 三、实验步骤及实验结果

除去程序输入格式控制的这一部分代码，此次实验主要的代码可分为下面几个实现：

- ping 程序。
- TCP connect 扫描程序；
- TCP SYN 扫描程序；
- TCP FIN 扫描程序；
- UDP 扫描程序；

实验中我们需要创建端口扫描器，然后根据用户输入来调用相应的模块，流程图如下：



## （一）端口扫描器头文件

为使端口扫描器功能模块化、可复用，定义了一个头文件用于声明必要的数据结构 and 函数声明：

### 1. 结构体

- IPHeader：定义了 IP 头部的结构体，用于构造 IP 数据包。

```

// IP头部结构体
struct IPHeader {
    unsigned char headerLen : 4; // 头部长度（单位：32位字节），占4位
    unsigned char version : 4; // 版本号，IPv4或IPv6，占4位
    unsigned char tos; // 服务类型，占8位
    unsigned short length; // 总长度，占16位
    unsigned short ident; // 标识，占16位
    unsigned short fragFlags; // 分片偏移和标志位，占16位
    unsigned char ttl; // 存活时间（TTL），占8位
    unsigned char protocol; // 协议类型，占8位
    unsigned short checksum; // 校验和，占16位
    unsigned int srcIP; // 源IP地址，占32位
    unsigned int dstIP; // 目的IP地址，占32位

    // 初始化
    IPHeader(unsigned int src, unsigned int dst, int protocol) {
        version = 4; // 设置版本号为IPv4
        headerLen = 5; // 设置头部长度为5个32位字节（20字节）
        srcIP = src; // 设置源IP地址
        dstIP = dst; // 设置目的IP地址
        ttl = (char)128; // 设置存活时间（TTL）为128
        this->protocol = protocol; // 设置协议类型
        if (protocol == IPPROTO_TCP) {
            length = htons(20 + 20); // 如果协议类型为TCP，则设置总长度为20字节
        } else if (protocol == IPPROTO_UDP) {
            length = htons(20 + 8); // 如果协议类型为UDP，则设置总长度为20字节
        }
    }
};

```

- TCPHeader: 定义了 TCP 头部的结构体，用于构造 TCP 数据包。

```
// TCP头部结构体
struct TCPHeader {
    uint16_t srcPort;    // 源端口号, 占16位
    uint16_t dstPort;    // 目的端口号, 占16位
    uint32_t seq;        // 序列号, 占32位
    uint32_t ack;        // 确认号, 占32位
    uint8_t null1 : 4;   // 未使用的字段, 占4位
    uint8_t length : 4;  // 数据偏移, 占4位
    uint8_t FIN : 1;     // FIN标志位
    uint8_t SYN : 1;     // SYN标志位
    uint8_t RST : 1;     // RST标志位
    uint8_t PSH : 1;     // PSH标志位
    uint8_t ACK : 1;     // ACK标志位
    uint8_t URG : 1;     // URG标志位
    uint8_t null2 : 2;   // 未使用的字段
    uint16_t windowSize; // 窗口大小
    uint16_t checksum;   // 校验和
    uint16_t ptr;        // 紧急指针
};
```

- pseudohdr: 定义了 TCP 伪头部的结构体, 用于计算 TCP 校验和。

```
//
struct pseudohdr
{
    unsigned int saddr;    // 源IP地址
    unsigned int daddr;    // 目的IP地址
    char useless;          // 未使用的字段, 填充字节
    unsigned char protocol; // 协议类型 (TCP或UDP)
    unsigned short length; // TCP或UDP数据长度
};
```

- 各扫描线程参数的结构体。

## 2. 函数声明:

- Ping: 用于发送 ICMP 报文对指定主机进行探测, 检查主机的存活性。
- 各模块扫描线程函数。

## 3. 辅助函数:

- in\_cksum: 用于计算校验和的辅助函数。

```

/*
static inline unsigned short in_cksum(unsigned short *ptr, int nbytes)
{
    register long sum;          // 校验和
    u_short oddbyte;           // 不足16位的字节
    register u_short answer;    // 最终的校验和结果

    sum = 0;                    // 初始化校验和为0
    while(nbytes > 1)           // 遍历数据的每个16位字
    {
        sum += *ptr++;          // 将每个16位字累加到校验和中
        nbytes -= 2;            // 字节数减去2, 因为每次操作都是处理两个字节
    }

    if(nbytes == 1)             // 如果数据长度为奇数, 则处理最后一个不足16位的字节
    {
        oddbyte = 0;            // 初始化不足16位的字节为0
        *((u_char *) &oddbyte) = *(u_char *)ptr; // 将最后一个字节复制到oddbyte中
        sum += oddbyte;         // 将oddbyte加到校验和中
    }

    sum = (sum >> 16) + (sum & 0xffff); // 将校验和的高16位加到低16位上, 并清除溢出的高位
    sum += (sum >> 16);           // 将上一步的溢出值加到低16位上
    answer = ~sum;               // 取反得到最终的校验和结果

    return(answer);              // 返回校验和结果
}

```

- GetLocalHostIP: 用于获取本地主机的 IP 地址。

```

/*
static inline unsigned int GetLocalHostIP(void)
{
    FILE *fd;                  // 文件指针
    char buf[20] = {0x00};     // 用于存储命令输出结果的缓冲区

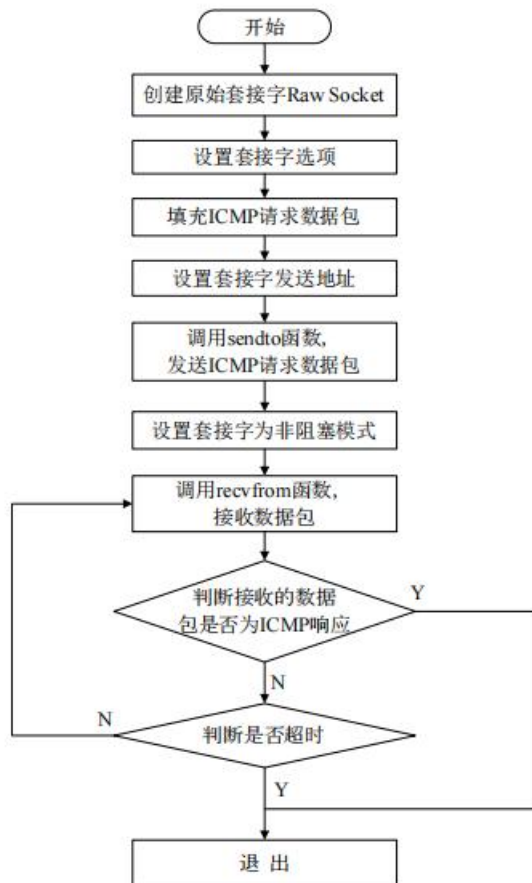
    // 执行系统命令并打开文件流, 获取本地主机的IP地址信息
    fd = popen("/sbin/ifconfig | grep inet | grep -v 127 | awk '{print $2}' | cut -d \":\"";
    if(fd == NULL)
    {
        fprintf(stderr, "cannot get source ip -> use the -f option\n"); // 输出错误信息
        exit(-1); // 退出程序
    }
    fscanf(fd, "%20s", buf);    // 从文件流中读取IP地址信息
    return(inet_addr(buf));     // 将字符串形式的IP地址转换为网络字节序的整数形式并返回
}

```

## (二) ICMP 探测指定主机

Ping 程序流程如下:





首先我们创建 ICMP 套接字：

```

/*
bool Ping(std::string HostIP, unsigned LocalHostIP) {
    struct iphdr *ip;           // IP 头部指针
    struct icmphdr *icmp;       // ICMP 头部指针
    unsigned short LocalPort = 8888; // 本地端口号
    int PingSock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP); // 创建 ICMP 套接字

    // 检查套接字是否创建成功
    if(PingSock < 0) {
        std::cout << "socket error" << std::endl;
        return false;
    }
}

```

然后我们创建 ICMP 数据包并填充：

```

int SendBufSize = sizeof(struct iphdr) + sizeof(struct icmphdr) + sizeof(struct timeval); //
char *SendBuf = (char*)malloc(SendBufSize); // 分配发送缓冲区内存
memset(SendBuf, 0, sizeof(SendBuf)); // 清空发送缓冲区

ip = (struct iphdr*)SendBuf; // IP 头部指针指向发送缓冲区
// 填充 IP 头部
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 头部
icmp = (struct icmphdr*)(ip + 1);
icmp->type = ICMP_ECHO;
icmp->code = 0;
icmp->un.echo.id = htons(LocalPort);
icmp->un.echo.sequence = 0;

```

接下来我们向目标主机发送 ICMP 报文：

```

// 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, Addrlen);
if(ret < 0) {
    std::cout << "sendto error" << std::endl;
    return false;
}

// 设置套接字为非阻塞模式
if(fcntl(PingSock, F_SETFL, O_NONBLOCK) == -1) {
    perror("fcntl error");
    return false;
}

```

最后我们通过是否接收到 ICMP 响应报文并解析对比 ICMP 响应报文 IP 地址，

来判断目标主机是否可达：

```

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));
    if (TimeUse < 3) {
        continue;
    }
    else {
        flags = false;
        break;
    }
} while(true);
return flags;

```

### (三) TCP connect 扫描

TCP Connect 扫描时通过调用流套接字 (SOCK\_STREAM) 的 connect 函数实现的。

该函数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。

在编程中为了提高效率, 采用了创建子线程同时扫描目标主机多个端口的方法。

两个线程函数为: Thread\_TCPconnectScan 和 Thread\_TCPconnectHost。其中 Thread\_TCPconnectScan 是扫描的主线程函数, 该函数在扫描器的主流程中

被调用，用于遍历目标主机的端口，创建负责扫描某一固定端口的子线程。而连接(connect)目标主机指定端口的工作正是由线程函数 Thread\_TCPconnectHost 来完成的。

### 线程函数 Thread\_TCPconnectHost 的流程：

```
//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);
```

(1) 获得线程函数 Thread\_TCPconnectScan 传来的目标主机 IP 地址和扫描端口号。

(2) 创建流套接字 ConSock。

(3) 设置连接目标主机的套接字地址 HostAddr。

(4) 调用 connect 函数连接目标主机。若函数返回-1，则连接失败；否则，连接成功。

(5) 关闭套接字 ConSock，子线程数 TCPConThrdNum 减 1，退出线程。

## 线程函数 Thread\_TCPconnectScan 流程:

```
//获得扫描的目标主机IP, 起始端口, 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);
}
```

- (1) 首先从参数中获取目标主机的 IP 地址、起始端口和终止端口。
- (2) 然后创建多个线程来并行扫描这些端口。每个线程负责连接到目标主机的一个特定端口, 并根据连接成功与否来判断端口的状态(开放或关闭)。
- (3) 在创建每个连接线程时, 会为该线程分配一个`TCPConHostThrParam`结构体, 线程通过调用`Thread\_TCPconnectHost`函数来进行连接, 并处理连接结果。

- (4) 为了控制并发线程的数量，函数会维护一个`TCPConThrdNum`变量，用于记录当前活跃的线程数。
- (5) 等待所有连接线程都执行完毕，然后退出线程。

## **(四) TCP SYN 扫描**

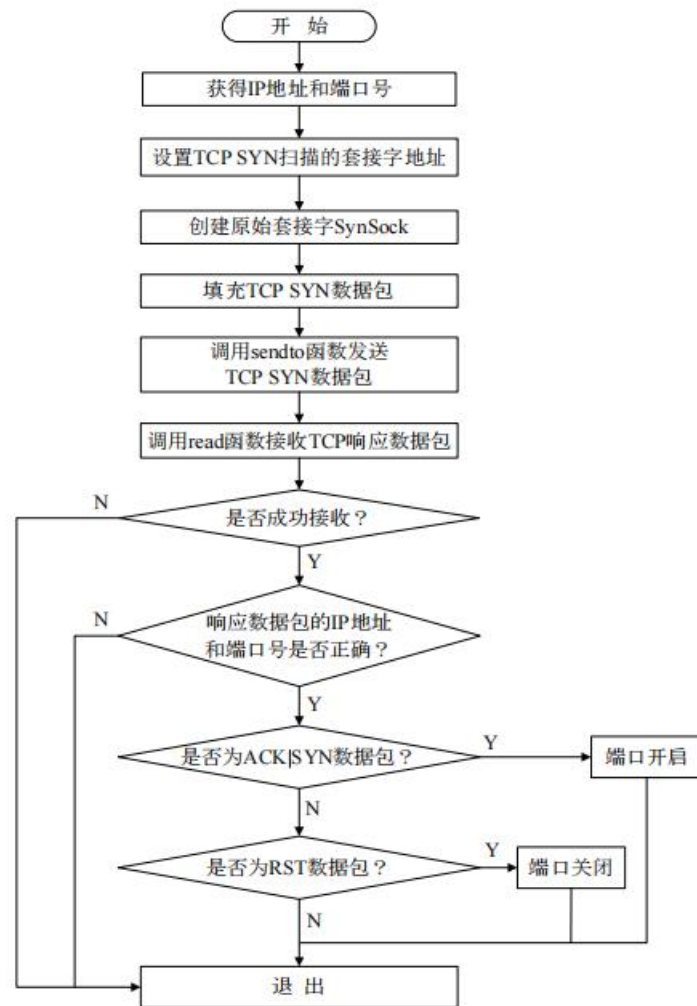
如同 TCP connect 扫描一样，TCP SYN 扫描也包含了两个线程函数：

Thread\_TCPSynScan 和 Thread\_TCPSYNHost。

Thread\_TCPSynScan 是主线程函数，负责遍历目标主机的被测端口，并调用 Thread\_TCPSYNHost 函数创建多个扫描子线程。Thread\_TCPSYNHost 函数用于完成对目标主机指定端口的 TCP SYN 扫描。



## 线程函数 Thread\_TCPSYNHost 流程:



代码如下:

(1) 设置目标主机信息, 创建套接字:

```

struct TCPSYNHostThrParam *p = (struct TCPSYNHostThrParam*) param;
std::string HostIP = p -> HostIP; // 目标主机的IP地址
unsigned HostPort = p -> HostPort; // 目标主机的端口号
unsigned LocalPort = p -> LocalPort; // 本地主机的端口号
unsigned LocalHostIP = p -> LocalHostIP; // 本地主机的IP地址

```

```

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);

```

```

// 创建原始套接字
int SynSock = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
if(SynSock < 0) {
    perror("Can't create raw socket !");
    pthread_exit(NULL);
}

```

(2) 填充报文信息:

```

// 填充TCP SYN数据包头部信息
char sendbuf[sizeof(struct pseudohdr) + sizeof(struct tcphdr)];
struct pseudohdr *ptcph = (struct pseudohdr*) sendbuf;
struct tcphdr *tcph = (struct tcphdr*)(sendbuf + sizeof(struct pseudohdr));
ptcph -> saddr = LocalHostIP; // 本地IP地址
ptcph -> daddr = inet_addr(HostIP.c_str()); // 目标IP地址
ptcph -> useless = 0; // 无用字段
ptcph -> protocol = IPPROTO_TCP; // 协议类型
ptcph -> length = htons(sizeof(struct tcphdr)); // TCP头长度

// 填充TCP头部信息
memset(tcph, 0, sizeof(struct tcphdr));
tcph->th_sport = htons(LocalPort); // 本地端口号
tcph->th_dport = htons(HostPort); // 目标端口号
tcph->th_seq = htonl(123456); // 序列号
tcph->th_ack = 0; // 确认号
tcph->th_off = 5; // 数据偏移
tcph->th_flags = TH_SYN; // TCP SYN标志
tcph->th_win = htons(65535); // 窗口大小
tcph->th_sum = 0; // 校验和
tcph->th_urp = 0; // 紧急指针
tcph->th_sum = in_cksum((unsigned short*)ptcph, sizeof(struct pseudohdr) + sizeof(struct tcphdr));

// 构造IP头部信息
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));

```

(3) 发送数据包:



```

// 发送TCP SYN数据包
int len = sendto(SynSock, temp, sizeof(IPHeader) + sizeof(struct tcphdr), 0, (struct sockaddr *)&SYNScanHostAc
if(len < 0) {
    perror("Send TCP SYN Packet error");
    close(SynSock);
    pthread_exit(NULL);
}

```

#### (4) 解析响应数据包:

```

// 接收并解析响应数据包
struct ip *iph;
do {
    len = recvfrom(SynSock, recvbuf, sizeof(recvbuf), 0, NULL, NULL);
    if(len < 0) {
        perror("Read TCP SYN Packet error");
        close(SynSock);
        pthread_exit(NULL);
    }
    else {
        iph = (struct ip *)recvbuf;
        int i = iph -> ip_hl * 4;
        tcph = (struct tcphdr *) (recvbuf + i);

        // 解析IP和TCP头信息
        std::string SrcIP = inet_ntoa(iph -> ip_src);
        std::string DstIP = inet_ntoa(iph -> ip_dst);
        unsigned SrcPort = ntohs(tcph -> th_sport);
        unsigned DstPort = ntohs(tcph -> th_dport);

        // 判断端口状态并输出结果
        if(HostIP == SrcIP && LocalIP == DstIP && SrcPort == HostPort && DstPort == LocalPort) {
            if(tcph->th_flags == 0x12) { // SYN|ACK数据包
                pthread_mutex_lock(&TCPSynPrintlocker);
                std::cout << "Host: " << SrcIP << " Port: " << ntohs(tcph -> th_sport) << " open !"
                pthread_mutex_unlock(&TCPSynPrintlocker);
            }
            if(tcph->th_flags == 0x14) { // RST数据包
                pthread_mutex_lock(&TCPSynPrintlocker);
                std::cout << " Port: " << ntohs(tcph -> th_sport) << " closed !" << std::endl;
                pthread_mutex_unlock(&TCPSynPrintlocker);
            }
        }
    }
}

```

## 线程函数 Thread\_TCPSynScan 流程:

```
//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) {
```

- (1) 获取目标主机的 IP 地址、起始端口和终止端口。
- (2) 然后创建多个线程来并行扫描这些端口。
- (3) 维护一个`TCPSynThrdNum`变量，用于记录当前活跃的线程数。如果如果子线程数大于 100，则暂时休眠。
- (4) 等待所有连接线程都执行完毕，然后退出线程。

## （五）TCP FIN 扫描

TCP FIN 扫描的代码与 TCP SYN 扫描的代码基本相同。

都利用原始套接字构造 TCP 数据包发送给目标主机的被测端口。不同的只是将 TCP 头 flags 字段的 FIN 位置 1。另外在接收 TCP 响应数据包时也略有不同。

和前面两种扫描一样，TCP FIN 扫描也由两个线程函数构成。它们分别是 Thread\_TCPFinScan 和 Thread\_TCPFINHost。

### 线程函数 Thread\_TCPFINHost 流程：

- （1）获得目标主机地址和端口号，创建 FIN 发送数据包和接收数据包。

```

void* Thread_TCPFINHost(void* param) {
    // 填充 TCP FIN 数据包
    struct TCPFINHostThrParam *p = (struct TCPFINHostThrParam*)param;
    std::string HostIP = p->HostIP; // 目标主机 IP 地址
    unsigned HostPort = p->HostPort; // 目标主机端口号
    unsigned LocalPort = p->LocalPort; // 本地端口号
    unsigned LocalHostIP = p->LocalHostIP; // 本地主机 IP 地址

    // 填充目标主机地址结构体
    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);

    // 创建原始套接字用于发送 TCP FIN 包
    int FinSock = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
    if (FinSock < 0) {
        pthread_mutex_lock(&TCPFinPrintlocker);
        std::cout << "Can't create raw socket !" << std::endl;
        pthread_mutex_unlock(&TCPFinPrintlocker);
    }

    // 创建原始套接字用于接收 TCP FIN 包
    int FinRevSock = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
    if (FinRevSock < 0) {
        pthread_mutex_lock(&TCPFinPrintlocker);
        std::cout << "Can't create raw socket !" << std::endl;
        pthread_mutex_unlock(&TCPFinPrintlocker);
    }

    // 设置套接字选项，用于构建 IP 数据包头部
    int flag = 1;

```

(2) 填充、发送数据包，并将另一个套接字设置为非阻塞模式进行接收

```

// 构建 IP 数据包头部
IPHeader IPHeader(ptcp->saddr, ptcp->daddr, IPPROTO_TCP);
char temp[sizeof(IPHeader) + sizeof(struct tcphdr)];
memcpy((void*)temp, (void*)&IPHeader, sizeof(IPHeader));
memcpy((void*)(temp+sizeof(IPHeader)), (void*)tcp, sizeof(struct tcphdr));

// 发送 TCP FIN 数据包
int len = sendto(FinSock, temp, sizeof(IPHeader) + sizeof(struct tcphdr), 0, (struct sockaddr*)&FinAddr, sizeof(FinAddr));
if(len < 0) {
    pthread_mutex_lock(&TCPFinPrintlocker);
    std::cout << "Send TCP FIN Packet error !" << std::endl;
    pthread_mutex_unlock(&TCPFinPrintlocker);
}

// 将接收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);
}

```

### (3) 解析接收数据包并判断

```

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 *tcp = (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(tcp->th_sport);
            unsigned DstPort = ntohs(tcp->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, 目的端口是否等于目标端口
            if(HostIP == SrcIP && LocalIP == DstIP && SrcPort == HostPort && DstPort == LocalPort) {
                // 判断是否为 RST 数据包
                if (tcp->th_flags == 0x14) {
                    pthread_mutex_lock(&TCPFinPrintlocker);
                    std::cout << "Host: " << SrcIP << " Port: " << ntohs(tcp->th_sport) << " closed !" << std::endl;
                    pthread_mutex_unlock(&TCPFinPrintlocker);
                }
                break;
            }
        }
    }
} while(1);

// 判断等待响应数据包时间是否超过 5 秒
gettimeofday(&TpEnd, NULL);
float Timeout = (1000000 * (TpEnd.tv_sec - Tstart.tv_sec) + (TpEnd.tv_usec - Tstart.tv_usec)) / 1000000;

```



## 线程函数 Thread\_TCPFinScan 流程:

```
// 线程函数，用于执行 TCP FIN 扫描
void* Thread_TCPFinScan(void* param) {
    // 获取参数结构体
    struct TCPFINThrParam *p = (struct TCPFINThrParam*)param;
    // 提取参数信息
    std::string HostIP = p->HostIP;
    unsigned BeginPort = p->BeginPort;
    unsigned EndPort = p->EndPort;
    unsigned LocalHostIP = p->LocalHostIP;

    // 初始化线程计数器和本地端口
    TCPFinThrdNum = 0;
    unsigned LocalPort = 1024;

    // 初始化线程属性和线程ID
    pthread_attr_t attr, lattr;
    pthread_t listenThreadID, subThreadID;

    // 遍历需要扫描的端口范围
    for (unsigned TempPort = BeginPort; TempPort <= EndPort; TempPort++) {
        // 创建 TCP FIN 扫描主机参数结构体
        struct TCPFINHostThrParam *pTCPFINHostParam = new TCPFINHostThrParam;
        pTCPFINHostParam->HostIP = HostIP;
        pTCPFINHostParam->HostPort = TempPort;
        pTCPFINHostParam->LocalPort = TempPort + LocalPort;
        pTCPFINHostParam->LocalHostIP = LocalHostIP;

        // 初始化线程属性
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

        // 创建子线程执行 TCP FIN 扫描
        int ret = pthread_create(&subThreadID, &attr, Thread_TCPFINHost, pTCP
        if (ret == -1) {
```

Thread\_TCPFinScan 线程函数和 TCP SYN 扫描的 Thread\_TCPSynSca 线程函数类似，故不在赘述。

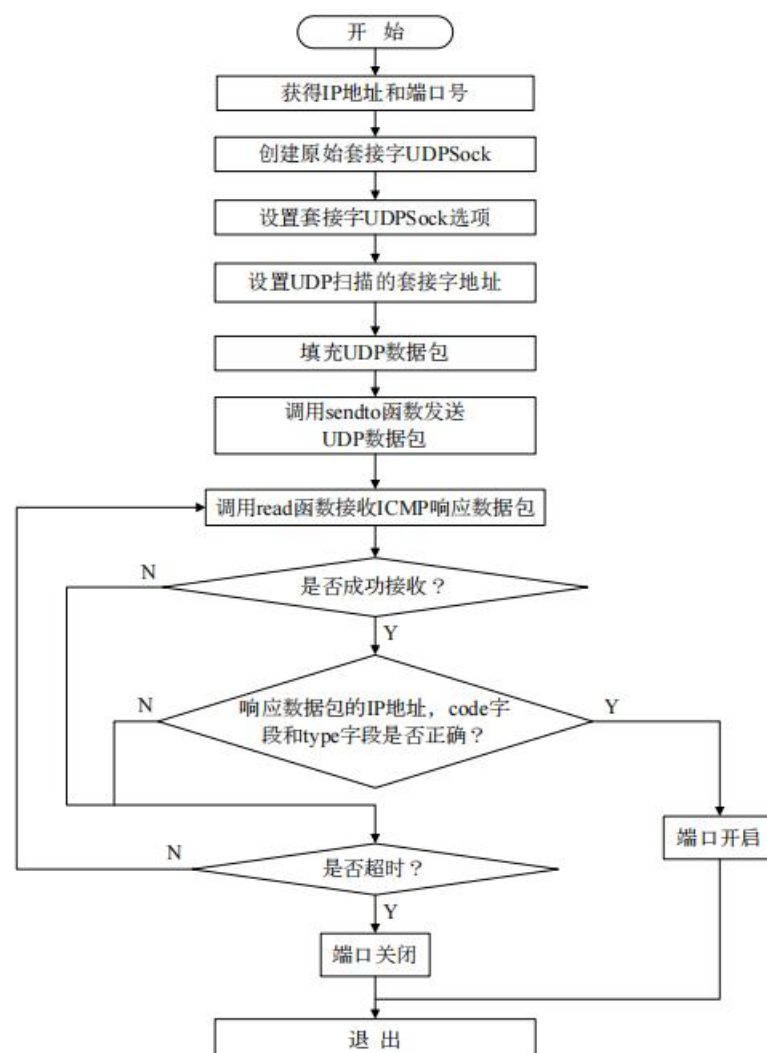
## (六) UDP 扫描

UDP 扫描是通过线程函数 Thread\_UDPScan 和普通函数 UDPScanHost 实现的。

UDP 扫描没有采用创建多个子线程同时扫描多个端口的方式，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。

线程函数 Thread\_UDPScan 负责遍历目标主机端口，调用函数 UDPScanHost 对指定端口进行扫描。

### 普通函数 UDPScanHost 流程：



(1) 获得目标主机 IP 地址和扫描端口号，以及本机 IP 地址和端口号。

```

// UDP 扫描主机函数
void* UDPScanHost(void* param) {
    // 获取参数结构体
    struct UDPScanHostThrParam *p = (struct UDPScanHostThrParam*)param;
    // 提取参数信息
    std::string HostIP = p->HostIP;
    unsigned HostPort = p->HostPort;
    unsigned LocalPort = p->LocalPort;
    unsigned LocalHostIP = p->LocalHostIP;
}

```

(2) 创建原始套接字 UDPSock。

```

// 创建 UDP 原始套接字
int UDPSock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
if (UDPSock < 0) {
    pthread_mutex_lock(&UDPPrintlocker);
    std::cout << "Can't create raw ICMP socket !" << std::endl;
    pthread_mutex_unlock(&UDPPrintlocker);
}

```

(3) 设置套接字 UDPSock 的选项。

```

// 设置套接字选项
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);
}

```

(4) 设置 UDP 扫描的套接字地址。

```

// 设置 UDP 扫描主机地址结构体
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);

```

(5) 填充 UDP 数据包。



```

// 构造 UDP 数据包
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(s

// 设置 UDP 头部字段
udp->source = htons(LocalPort);
udp->dest = htons(HostPort);
udp->len = htons(sizeof(struct udphdr));
udp->check = 0;

// 设置伪首部
pseudo->saddr = LocalHostIP;
pseudo->daddr = inet_addr(&HostIP[0]);
pseudo->useless = 0;
pseudo->protocol = IPPROTO_UDP;
pseudo->length = udp->len;

// 计算校验和
udp->check = in_cksum((u_short *)pseudo, sizeof(struct udphdr) + sizeof(struct pseudohdr

// 设置 IP 头部字段
ip->ihl = 5;
ip->version = 4;
ip->tos = 0x10;
ip->tot_len = sizeof(packet);
ip->frag_off = 0;

```

(6) 调用 sendto 函数向目标主机的指定端口发送 UDP 数据包。

```

// 发送 UDP 数据包
int n = sendto(UDPSock, packet, ip->tot_len, 0, (struct sockaddr *)&UDPScanHostAddr, sizeof(L
if (n < 0) {
    pthread_mutex_lock(&UDPPrintlocker);
    std::cout << "Send message to Host Failed !" << std::endl;
    pthread_mutex_unlock(&UDPPrintlocker);
}

```

(7) 调用 read 函数接收目标主机的 ICMP 响应数据包。

如果该响应数据包的源地址等于目标主机地址,code 字段和 type 字段的值都为 3,那么该数据包就是目标主机被扫描端口返回的 ICMP 不可达数据包。因此,可以认为被扫描的 UDP 端口是关闭的。

若接收时间超过 3 秒,则认为被扫描的 UDP 端口开启,退出循环。

```

// 接收 ICMP 响应数据包
struct timeval TpStart, TpEnd;
struct ipicmphdr hdr;
gettimeofday(&TpStart, NULL); // 获取开始时间
do {
    // 接收响应消息
    n = read(UDPSock, (struct ipicmphdr *)&hdr, sizeof(hdr));

    if (n > 0) {
        // 判断是否为 ICMP 目的不可达消息
        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;
        }
    }
}
// 判断是否超时
gettimeofday(&TpEnd, NULL);
float TimeUse = (1000000 * (TpEnd.tv_sec - TpStart.tv_sec) + (TpEnd.tv_usec - TpStart.tv_usec));
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);

```

(8) 关闭套接字 UDPSock, 返回。

## 线程函数 Thread\_UDPSca 流程:

在从起始端口 (BeginPort) 到终止端口 (EndPort) 的遍历中, 逐次对当前端口 (TempPort) 进行 UDP 扫描。

```

void* Thread_UDPScan(void* param) {
    // 获取参数结构体
    struct UDPThrParam *p = (struct UDPThrParam*)param;
    std::string HostIP = p->HostIP;
    unsigned BeginPort = p->BeginPort;
    unsigned EndPort = p->EndPort;
    unsigned LocalHostIP = p->LocalHostIP;

    // 设置本地端口起始值
    unsigned LocalPort = 1024;

    // 遍历需要扫描的端口范围
    for (unsigned TempPort = BeginPort; TempPort <= EndPort; TempPort++) {
        // 创建 UDP 扫描主机参数结构体
        UDPScanHostThrParam *pUDPScanHostParam = new UDPScanHostThrParam;
        pUDPScanHostParam->HostIP = HostIP;
        pUDPScanHostParam->HostPort = TempPort;
        pUDPScanHostParam->LocalPort = TempPort + LocalPort;
        pUDPScanHostParam->LocalHostIP = LocalHostIP;
        UDPScanHost(pUDPScanHostParam);
    }

    // 扫描线程退出消息
    std::cout << "UDP Scan thread exit !" << std::endl;
    pthread_exit(NULL);
}

```

## (七) 根据程序输入格式实现端口扫描器

```

int main(int argc, char *argv[]) {
    // 定义操作和对应函数的映射关系
    std::unordered_map<std::string, void(*)(int, char*> mapOp = {
        {"-h", print_h},
        {"-c", print_c},
        {"-s", print_s},
        {"-u", print_u},
        {"-f", print_f}
    };
}

```

其中 print\_h 打印帮助信息，print\_c 为 TCP connect 扫描，以此类推。

```

void print_c(int argc, char *argv[]) {
    // 显示开始 TCP 连接扫描信息
    std::cout << "Begin TCP connect scan..." << std::endl;

    // 设置 TCP 连接扫描参数
    TCPConParam.HostIP = HostIP;
    TCPConParam.BeginPort = BeginPort;
    TCPConParam.EndPort = EndPort;

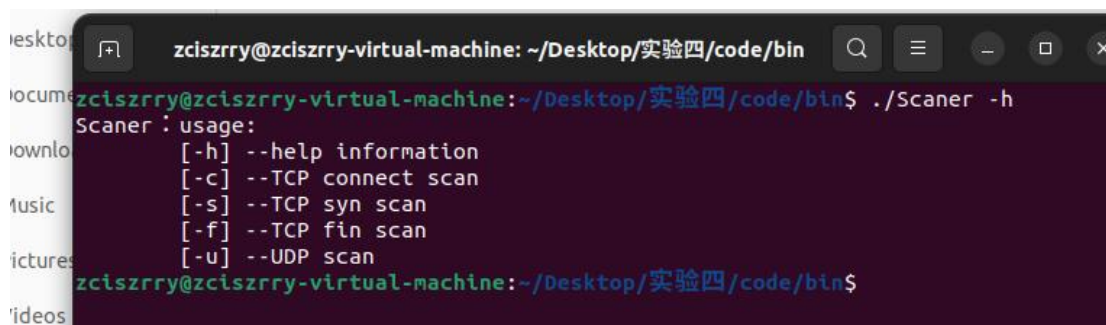
    // 创建 TCP 连接扫描线程
    int ret = pthread_create(&ThreadID, NULL, Thread_TCPconnectScan, &TCPConParam);
    if (ret == -1) {
        std::cout << "Can't create the TCP connect scan thread !" << std::endl;
        return;
    }

    // 等待 TCP 连接扫描线程结束
    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;
    }
}
}

```

## (八) 实验结果

打印帮助信息：



```

zciszrry@zciszrry-virtual-machine: ~/Desktop/实验四/code/bin
zciszrry@zciszrry-virtual-machine:~/Desktop/实验四/code/bin$ ./Scanner -h
Scanner : usage:
[-h] --help information
[-c] --TCP connect scan
[-s] --TCP syn scan
[-f] --TCP fin scan
[-u] --UDP scan
zciszrry@zciszrry-virtual-machine:~/Desktop/实验四/code/bin$

```

TCP connect 扫描



```

zciszrry@zciszrry-virtual-machine:~/Desktop/实验四/code/bin$ sudo su
root@zciszrry-virtual-machine:/home/zciszrry/Desktop/实验四/code/bin# ./Scanner -c
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: 127.0.0.1:5 is closed
TCP connect scan: 127.0.0.1:4 is closed
TCP connect scan: 127.0.0.1:2 is closed
TCP connect scan: 127.0.0.1:17 is closed
TCP connect scan: 127.0.0.1:3 is closed
TCP connect scan: 127.0.0.1:21 is closed
TCP connect scan: 127.0.0.1:27 is closed
TCP connect scan: 127.0.0.1:28 is closed
TCP connect scan: 127.0.0.1:11 is closed
TCP connect scan: 127.0.0.1:14 is closed
TCP connect scan: 127.0.0.1:8 is closed

```

```

TCP connect scan: 127.0.0.1:23 is closed
TCP connect scan: 127.0.0.1:57 is closed
TCP connect scan: 127.0.0.1:22 is open
TCP connect scan: 127.0.0.1:24 is closed
TCP connect scan: 127.0.0.1:16 is closed

```

## TCP SYN 扫描

```

root@zciszrry-virtual-machine:/home/zciszrry/Desktop/实验四/code/bin#
root@zciszrry-virtual-machine:/home/zciszrry/Desktop/实验四/code/bin# ./Scanner -s
Please input IP address of a Host:192.168.137.134
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 192.168.137.134 port 1~255 ...
Ping Host 192.168.137.134 Successfully !
Begin TCP SYN scan...
Port: 4 closed !
Port: 5 closed !
Port: 15 closed !
Port: 9 closed !
Port: 11 closed !
Port: 13 closed !
Port: 38 closed !
Port: 39 closed !
Port: 40 closed !
Port: 41 closed !

```

## TCP FIN 扫描

```

Port: 215 closed !
TCP SYN Scan finished !
root@zciszrry-virtual-machine:/home/zciszrry/Desktop/实验四/code/bin# ./Scanner -f
Please input IP address of a Host:192.168.137.134
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 192.168.137.134 port 1~255 ...
Ping Host 192.168.137.134 Successfully !
Begin TCP FIN scan...
Host: 192.168.137.134 Port: 3 closed !
Host: 192.168.137.134 Port: 4 closed !
Host: 192.168.137.134 Port: 11 closed !
Host: 192.168.137.134 Port: 1 closed !
Host: 192.168.137.134 Port: 13 closed !
Host: 192.168.137.134 Port: 14 closed !
Host: 192.168.137.134 Port: 15 closed !
Host: 192.168.137.134 Port: 17 closed !

```

## UDP 扫描

```

[sudo] password for zciszrry:
root@zciszrry-virtual-machine:/home/zciszrry/Desktop/实验四/code/bin# ./Scanner -u
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 closed !
Host: 127.0.0.1 Port: 2 closed !
Host: 127.0.0.1 Port: 3 closed !

```

## 四、实验遇到的问题及其解决方法

实验中,进行停用 iptables 服务时报错,尝试检查 iptables 服务状态时,却提示找不到 iptables.service 单元,可能是系统使用了不同的防火墙管理机制。

再次阅读文档,发现主要原因很可能是我没有用 root 权限运行程序,于是添加 root 授权步骤:

```

zciszrry@zciszrry-virtual-machine:~/Desktop/实验四/code/bin$
zciszrry@zciszrry-virtual-machine:~/Desktop/实验四/code/bin$
zciszrry@zciszrry-virtual-machine:~/Desktop/实验四/code/bin$ sudo su
root@zciszrry-virtual-machine:/home/zciszrry/Desktop/实验四/code/bin# ./Scanner -c
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 !

```

## 五、实验结论

在本次实验中，我学习到了端口扫描器这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。

通过实验，还学习到 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。

另外，在实验程序编写中，复习并进一步熟悉了 Linux 系统的套接字编程技术，学习了多线程编程的基本方法。

通过这几次网络安全实验，学习到了很多网络安全相关的知识，受益匪浅。