



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

网络安全技术作业报告

端口扫描器的设计与实现

朱浩泽 1911530

年级：2019 级

专业：计算机科学与技术

班级：计算机科学与技术 2 班

2022 年 6 月 10 日

目录

一、 实验介绍	1
(一) 实验目的	1
(二) 实验流程	1
二、 实验步骤	1
(一) 实验环境	1
(二) 核心代码实现	1
1. 代码框架	1
2. IP 头、TCP 头和 TCP 伪头以及一些工具函数	2
3. ICMP 探测指定主机	4
4. TCP connect 扫描	6
5. TCP SYN 扫描	8
6. TCP FIN 扫描	13
7. UDP 扫描	18
8. 端口扫描器程序 Scanner	22
三、 实验结论	22

一、 实验介绍

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。

(一) 实验目的

- 掌握端口扫描器的基本设计方法。
- 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。
- 熟练掌握 Linux 环境下的套接字编程技术。
- 掌握 Linux 环境下多线程编程的基本方法

(二) 实验流程

- 编写端口扫描程序，提供 TCP connect 扫描
- 编写端口扫描程序，提供 TCP SYN 扫描
- 编写端口扫描程序，提供 TCP FIN 扫描
- 编写端口扫描程序，提供 UDP 扫描
- 设计并实现 ping 程序，探测目标主机是否可达。

二、 实验步骤

(一) 实验环境

Ubuntu Server 20.04 LTS 64bit (腾讯云服务器), C++11, Cmake

(二) 核心代码实现

1. 代码框架

```
1 |— include
2 |   |— header.h
3 |— src
4 |   |— TCPConnectScan.cpp
5 |   |— TCPFINScan.cpp
6 |   |— TCPSYNScan.cpp
7 |   |— UDPScan.cpp
8 |   |— main.cpp
```

header.h 中包含了端口扫描器的基本结构体和函数声明。TCPConnectScan.cpp、TCPFINScan.cpp、TCPSYNScan.cpp、UDPScan.cpp 和 main.cpp 中分别实现了 TCP connect 扫描、TCP FIN 扫描、TCP SYN 扫描、UDP 扫描以及 ping 程序的实现。

2. 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

```
1 struct TCPHeader {
2     uint16_t srcPort;
3     uint16_t dstPort;
4     uint32_t seq;
5     uint32_t ack;
6     uint8_t null1 : 4;
7     uint8_t length : 4;
8     uint8_t FIN : 1;
9     uint8_t SYN : 1;
10    uint8_t RST : 1;
11    uint8_t PSH : 1;
12    uint8_t ACK : 1;
13    uint8_t URG : 1;
14    uint8_t null2 : 2;
15    uint16_t windowSize;
16    uint16_t checksum;
17    uint16_t ptr;
18 };
```

TCP 伪头，用于计算 TCP 头的校验和

```
1 struct pseudohdr
2 {
3     unsigned int saddr;
4     unsigned int daddr;
5     char useless;
6     unsigned char protocol;
7     unsigned short length;
8 };
```

IP 头，用于发送 IP 报文

```
1 struct IPHeader {
2     unsigned char headerLen : 4;
3     unsigned char version : 4;
4     unsigned char tos;
5     unsigned short length;
6     unsigned short ident;
7     unsigned short fragFlags;
8     unsigned char ttl;
9     unsigned char protocol;
10    unsigned short checksum;
11    unsigned int srcIP;
12    unsigned int dstIP;
13    IPHeader(unsigned int src, unsigned int dst, int protocol) {
14        version = 4;
15        headerLen = 5;
16        srcIP = src;
17        dstIP = dst;
18        ttl = (char)128;
19        this->protocol = protocol;
20        if (protocol == IPPROTO_TCP) {
21            length = htons(20 + 20);
```

```

22     }
23     else if (protocol == IPPROTO_UDP) {
24         length = htons(20 + 8);
25     }
26 }
27 };

```

校验和计算函数

```

1 static inline unsigned short in_cksum(unsigned short *ptr, int nbytes)
2 {
3     register long sum;
4     u_short oddbyte;
5     register u_short answer;
6
7     sum = 0;
8     while(nbytes > 1)
9     {
10         sum += *ptr++;
11         nbytes -= 2;
12     }
13
14     if(nbytes == 1)
15     {
16         oddbyte = 0;
17         *((u_char *) &oddbyte) = *(u_char *)ptr;
18         sum += oddbyte;
19     }
20
21     sum = (sum >> 16) + (sum & 0xffff);
22     sum += (sum >> 16);
23     answer = ~sum;
24
25     return(answer);
26 }

```

获取本地 IP 地址

```

1 static inline unsigned int GetLocalHostIP(void)
2 {
3     FILE *fd;
4     char buf[20] = {0x00};
5
6     fd = popen("/sbin/ifconfig | grep inet | grep -v 127 | awk '{print $2}' | cut -d
7         \":\" -f 2", "r");
8     if(fd == NULL)
9     {
10         fprintf(stderr, "cannot get source ip -> use the -f option\n");
11         exit(-1);
12     }
13     fscanf(fd, "%20s", buf);
14     return(inet_addr(buf));
15 }

```

3. ICMP 探测指定主机

该程序用于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

```
1 bool Ping(std::string HostIP, unsigned LocalHostIP) {
2     struct iphdr *ip;
3     struct icmp_hdr *icmp;
4     unsigned short LocalPort = 8888;
5
6     int PingSock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
7
8     if(PingSock < 0) {
9         std::cout << "socket error" << std::endl;
10        return false;
11    }
12
13    int on = 1;
14    int ret = setsockopt(PingSock, 0, IP_HDRINCL, &on, sizeof(on));
15
16    if(ret < 0) {
17        std::cout << "setsockopt error" << std::endl;
18        return false;
19    }
```

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

```
1 int SendBufSize = sizeof(struct iphdr) + sizeof(struct icmp_hdr) + sizeof(struct
    timeval);
2 char *SendBuf = (char*)malloc(SendBufSize);
3 memset(SendBuf, 0, sizeof(SendBuf));
4
5 ip = (struct iphdr*)SendBuf;
6 ip->ihl = 5;
7 ip->version = 4;
8 ip->tos = 0;
9 ip->tot_len = htons(SendBufSize);
10 ip->id = rand();
11 ip->ttl = 64;
12 ip->frag_off = 0x40;
13 ip->protocol = IPPROTO_ICMP;
14 ip->check = 0;
15 ip->saddr = LocalHostIP;
16 ip->daddr = inet_addr(&HostIP[0]);
17
18 //填充icmp头
19 icmp = (struct icmp_hdr*)(ip + 1);
20 icmp->type = ICMP_ECHO;
21 icmp->code = 0;
22 icmp->un.echo.id = htons(LocalPort);
23 icmp->un.echo.sequence = 0;
24
25 struct timeval *tp = (struct timeval*) &SendBuf[28];
26 gettimeofday(tp, NULL);
27 icmp->checksum = in_cksum((u_short *)icmp, sizeof(struct icmp_hdr) + sizeof(struct
    timeval));
```

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。

```

1 //设置套接字的发送地址
2 struct sockaddr_in PingHostAddr;
3 PingHostAddr.sin_family = AF_INET;
4 PingHostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
5 int Addrlen = sizeof(struct sockaddr_in);
6
7 //发送ICMP请求
8 ret = sendto(PingSock, SendBuf, SendBufSize, 0, (struct sockaddr*) &PingHostAddr,
9             sizeof(PingHostAddr));
10 if(ret < 0) {
11     std::cout << "sendto error" << std::endl;
12     return false;
13 }
14
15 if(fcntl(PingSock, F_SETFL, O_NONBLOCK) == -1) {
16     perror("fcntl error");
17     return false;
18 }

```

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。

```

1 struct timeval TpStart, TpEnd;
2 bool flags;
3 //循环等待接收ICMP响应
4 gettimeofday(&TpStart, NULL); //获得循环起始时刻
5 flags = false;
6
7 char RecvBuf[1024];
8 struct sockaddr_in FromAddr;
9 struct icmp* Recvicmp;
10 struct ip* Recvip;
11 std::string SrcIP, DstIP, LocalIP;
12 struct in_addr in_LocalhostIP;
13
14 do {
15     //接收ICMP响应
16     ret = recvfrom(PingSock, RecvBuf, 1024, 0, (struct sockaddr*) &FromAddr,
17                 (socklen_t*) &Addrlen);
18     if (ret > 0) //如果接收到一个数据包，对其进行解析
19     {
20         Recvip = (struct ip*) RecvBuf;
21         Recvicmp = (struct icmp*) (RecvBuf + (Recvip -> ip_hl * 4));
22         SrcIP = inet_ntoa(Recvip -> ip_src); //获得响应数据包IP头的源地址
23         DstIP = inet_ntoa(Recvip -> ip_dst); //获得响应数据包IP头的目的地址
24         in_LocalhostIP.s_addr = LocalHostIP;
25         LocalIP = inet_ntoa(in_LocalhostIP); //获得本机IP地址
26         //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
27         //本机IP地址，ICMP头的type字段是否为ICMP_ECHOREPLY

```

```

28     if (SrcIP == HostIP && DstIP == LocalIP &&
29         Recvicmp->icmp_type == ICMP_ECHOREPLY) {
30         /*ping成功, 退出循环*/
31         std::cout << "Ping Host " << HostIP << " Successfully !" << std::endl;
32         flags = true;
33         break;
34     }
35 }
36 //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。
37 gettimeofday(&TpEnd, NULL);
38 float TimeUse = (1000000 * (TpEnd.tv_sec - TpStart.tv_sec) + (TpEnd.tv_usec -
39     TpStart.tv_usec)) / 1000000.0;
40 if(TimeUse < 3) {
41     continue;
42 }
43 else {
44     flags = false;
45     break;
46 }
47 } while(true);
48 return flags;
49 }

```

4. TCP connect 扫描

这一部分我们使用的数据结构如下:

```

1 struct TCPConHostThrParam
2 {
3     std::string HostIP;
4     unsigned HostPort;
5 };
6
7 struct TCPConThrParam
8 {
9     std::string HostIP;
10    unsigned BeginPort;
11    unsigned EndPort;
12 };

```

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。

```

1 int TCPConThrdNum;
2 pthread_mutex_t TCPConPrintlocker = PTHREAD_MUTEX_INITIALIZER;
3 pthread_mutex_t TCPConScanlocker = PTHREAD_MUTEX_INITIALIZER;

```

然后我们编写 void* Thread_TCPconnectHost(void* param) 函数, 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。

```

1 void* Thread_TCPconnectHost(void* param) {
2     /*变量定义*/

```



```

3 //获得目标主机的IP地址和扫描端口号
4 struct TCPConHostThrParam *p = (struct TCPConHostThrParam*) param;
5 std::string HostIP = p -> HostIP;
6 unsigned HostPort = p -> HostPort;
7 //创建流套接字
8 int ConSock = socket(AF_INET, SOCK_STREAM, 0);
9 if(ConSock < 0) {
10     pthread_mutex_lock(&TCPConPrintlocker);
11
12 }

```

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。

```

1 //设置连接主机地址
2 struct sockaddr_in HostAddr;
3 memset(&HostAddr, 0, sizeof(HostAddr));
4 HostAddr.sin_family = AF_INET;
5 HostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
6 HostAddr.sin_port = htons(HostPort);
7 //connect目标主机
8 int ret = connect(ConSock, (struct sockaddr*) &HostAddr, sizeof(HostAddr));
9 if(ret < 0) {
10     pthread_mutex_lock(&TCPConPrintlocker);
11     std::cout << "TCP connect scan: " << HostIP << ":" << HostPort << " is closed"
12         << std::endl;
13     pthread_mutex_unlock(&TCPConPrintlocker);
14 } else {
15     pthread_mutex_lock(&TCPConPrintlocker);
16     std::cout << "TCP connect scan: " << HostIP << ":" << HostPort << " is open" <<
17         std::endl;
18     pthread_mutex_unlock(&TCPConPrintlocker);
19 }

```

然后我们关闭套接字，释放线程锁，线程数量减一。

```

1 delete p;
2 close(ConSock); //关闭套接字
3 //子线程数减1
4 pthread_mutex_lock(&TCPConScanlocker);
5 TCPConThrdNum--;
6 pthread_mutex_unlock(&TCPConScanlocker);
7 } // TCP connect 扫描

```

然后，我们编写 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、起始端口、终止端口，然后将线程数设置为 0。

```

1 void* Thread_TCPconnectScan(void* param)
2 {
3     /*变量定义*/
4     //获得扫描的目标主机IP，起始端口，终止端口
5     struct TCPConThrParam *p = (struct TCPConThrParam*) param;
6     std::string HostIP = p -> HostIP;
7     unsigned BeginPort = p -> BeginPort;
8     unsigned EndPort = p -> EndPort;

```

```

9   TCPConThrdNum = 0; //将线程数设为0
10  //开始从起始端口到终止端口循环扫描目标主机的端口

```

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。

```

1  pthread_t subThreadID;
2  pthread_attr_t attr;
3  for (unsigned TempPort = BeginPort; TempPort <= EndPort; TempPort++)
4  {
5      //设置子线程参数
6      TCPConHostThrParam *pConHostParam = new TCPConHostThrParam;
7      pConHostParam->HostIP = HostIP;
8      pConHostParam->HostPort = TempPort;
9      //将子线程设为分离状态
10     pthread_attr_init(&attr);
11     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
12     //创建connect目标主机指定的端口子线程
13     int ret = pthread_create(&subThreadID, &attr, Thread_TCPconnectHost,
14                             pConHostParam);
15     if(ret == -1) {
16         std::cout << "Create TCP connect scan thread error!" << std::endl;
17     }
18     //线程数加1
19     pthread_mutex_lock(&TCPConScanlocker);
20     TCPConThrdNum++;
21     pthread_mutex_unlock(&TCPConScanlocker);
22     //如果子线程数大于100，暂时休眠
23     while (TCPConThrdNum>100) {
24         sleep(3);
25     }
26 }

```

最后，我们等待子线程数为 0，返回。

```

1  while (TCPConThrdNum != 0) {
2      sleep(1);
3  }
4  pthread_exit(NULL);
5  }

```

5. TCP SYN 扫描

这一部分我们使用的数据结构如下：

```

1  struct TCPSYNHostThrParam
2  {
3      std::string HostIP;
4      unsigned HostPort;
5      unsigned LocalPort;
6      unsigned LocalHostIP;
7  };
8

```

```

9 struct TCPSYNThrParam
10 {
11     std::string HostIP;
12     unsigned BeginPort;
13     unsigned EndPort;
14     unsigned LocalHostIP;
15 };

```

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，具体来收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。

```

1 pthread_mutex_t TCPSynPrintlocker = PTHREAD_MUTEX_INITIALIZER;
2 pthread_mutex_t TCPSynScanlocker = PTHREAD_MUTEX_INITIALIZER;
3
4 int TCPSynThrdNum;

```

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

```

1 void* Thread_TCPSYNHost(void* param) {
2     /*变量定义*/
3     //获得目标主机的IP地址和扫描端口号
4     struct TCPSYNHostThrParam *p = (struct TCPSYNHostThrParam*) param;
5     std::string HostIP = p -> HostIP;
6     unsigned HostPort = p -> HostPort;
7     unsigned LocalPort = p -> LocalPort;
8     unsigned LocalHostIP = p -> LocalHostIP;
9
10    struct sockaddr_in SYNScanHostAddr;
11    memset(&SYNScanHostAddr, 0, sizeof(SYNScanHostAddr));
12    SYNScanHostAddr.sin_family = AF_INET;
13    SYNScanHostAddr.sin_addr.s_addr = inet_addr(HostIP.c_str());
14    SYNScanHostAddr.sin_port = htons(HostPort);

```

然后我们创建套接字

```

1 int SynSock = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
2 if(SynSock < 0) {
3     pthread_mutex_lock(&TCPSynPrintlocker);
4     std::cout << "Can't creat raw socket !" << std::endl;
5     pthread_mutex_unlock(&TCPSynPrintlocker);
6 }
7 int flag = 1;
8 if (setsockopt(SynSock, IPPROTO_IP, IP_HDRINCL, (void*)&flag, sizeof(int)) ==
9     -1) {
10     std::cout << "set IP_HDRINCL error.\n";
11 }

```

填充 TCP SYN 数据包

```

1 char sendbuf[8192];
2 char recvbuf[8192];
3 struct pseudohdr *ptcph = (struct pseudohdr*) sendbuf;
4 struct tcphdr *tcph = (struct tcphdr*)(sendbuf + sizeof(struct pseudohdr));

```

填充 TCP 伪头部，用于计算校验和

```

1  ptcph -> saddr = LocalHostIP;
2  ptcph -> daddr = inet_addr(HostIP.c_str());
3  in_addr src, dst;
4  ptcph -> useless = 0;
5  ptcph -> protocol = IPPROTO_TCP;
6  ptcph -> length = htons(sizeof(struct tcphdr));
7
8  src.s_addr = ptcph -> saddr;
9  dst.s_addr = ptcph -> daddr;

```

填充 TCP 头

```

1  memset(tcph, 0, sizeof(struct tcphdr));
2  // std::cout<<LocalPort<<" "<<HostPort<<std::endl;
3  tcph->th_sport = htons(LocalPort);
4  tcph->th_dport = htons(HostPort);
5  tcph->th_seq = htonl(123456);
6  tcph->th_ack = 0;
7  tcph->th_x2 = 0;
8  tcph->th_off = 5;
9  tcph->th_flags = TH_SYN;
10 tcph->th_win = htons(65535);
11 tcph->th_sum = 0;
12 tcph->th_urp = 0;
13 tcph->th_sum = in_cksum((unsigned short*)ptcph, 20 + 12);

```

封装 IP 头

```

1  IPHeader IPHeader(ptcph -> saddr, ptcph -> daddr, IPPROTO_TCP);
2  char temp[sizeof(IPHeader) + sizeof(struct tcphdr)];
3
4  memcpy((void*)temp, (void*)&IPHeader, sizeof(IPHeader));
5  memcpy((void*)(temp+sizeof(IPHeader)), (void*)tcph, sizeof(struct tcphdr));

```

发送 TCP SYN 数据包

```

1  int len = sendto(SynSock, temp, sizeof(IPHeader) + sizeof(struct tcphdr), 0, (struct
   sockaddr *)&SYNScanHostAddr, sizeof(SYNScanHostAddr));
2  // std::cout << sizeof(IPHeader) <<" "<< sizeof(struct tcphdr)<<" "<<len << std::
   endl;
3  if(len < 0) {
4      pthread_mutex_lock(&TCPSynPrintlocker);
5      std::cout << "Send TCP SYN Packet error !" << std::endl;
6      pthread_mutex_unlock(&TCPSynPrintlocker);
7  }

```

开始利用一个循环循环接受包到 buffer 中。

```

1  int count = 0;
2  std::string SrcIP;
3  struct ip *iph;
4  flag = 1;
5  sockaddr_in recvAddr;
6  int addrLen = sizeof(recvAddr);
7  do{
8      len = recvfrom(SynSock, recvbuf, 8192, 0, (sockaddr*)&recvAddr,

```

```

9         (socklen_t*)&addrLen);
10     if(len < 0) {
11         /*接收错误*/
12         pthread_mutex_lock(&TCPSynPrintlocker);
13         std::cout << "Read TCP SYN Packet error !" << std::endl;
14         pthread_mutex_unlock(&TCPSynPrintlocker);
15     }

```

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

```

1     else {
2         struct ip *iph = (struct ip *)recvbuf;
3         int i = iph -> ip_hl * 4;
4         tcph = (struct tcphdr *)(recvbuf + i);
5
6         std::string SrcIP = inet_ntoa(iph -> ip_src);
7         std::string DstIP = inet_ntoa(iph -> ip_dst);
8         struct in_addr in_LocalhostIP;
9         in_LocalhostIP.s_addr = LocalHostIP;
10        std::string LocalIP = inet_ntoa(in_LocalhostIP);
11
12        unsigned SrcPort = ntohs(tcph -> th_sport);
13        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

```

1        // std::cout << "-----" << std::endl;
2
3        // std::cout << HostIP << ' ' << SrcIP << std::endl;
4        // std::cout << LocalIP << ' ' << DstIP << std::endl;
5        // std::cout << SrcPort << ' ' << HostPort << std::endl;
6        // std::cout << DstPort << ' ' << LocalPort << std::endl;
7        if(HostIP == SrcIP && LocalIP == DstIP && SrcPort == HostPort && DstPort ==
            LocalPort)
8        {

```

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。

```

1
2        // std::cout<<(int)(tcph->th_flags)<<std::endl;
3        if(tcph->th_flags == 0x12) //判断是否为SYN|ACK数据包
4        {
5            /*端口开启*/
6            flag = 0;
7            pthread_mutex_lock(&TCPSynPrintlocker);
8            std::cout << "Host: " << SrcIP << " Port: " << ntohs(tcph -> th_sport)
                << " open !" << std::endl;
9            pthread_mutex_unlock(&TCPSynPrintlocker);
10        }
11        if(tcph->th_flags == 0x14) //判断是否为RST数据包
12        {
13            /*端口关闭*/
14            flag = 0;
15            pthread_mutex_lock(&TCPSynPrintlocker);
16            std::cout << " Port: " << ntohs(tcph -> th_sport) << " closed !" << std
                ::endl;

```

```

17         pthread_mutex_unlock(&TCPSynPrintlocker);
18     }
19 }
20 }
21 } while(count++ < 20 && flag);

```

最后，我们等待子线程数为 0，返回。

```

1 //退出子线程
2 if(flag){
3     pthread_mutex_lock(&TCPSynPrintlocker);
4     std::cout << "Host: " << SrcIP << " Port: " << HostPort << " closed !" << std::
        endl;
5     pthread_mutex_unlock(&TCPSynPrintlocker);
6 }
7 delete p;
8 close(SynSock);
9 pthread_mutex_lock(&TCPSynScanlocker);
10 TCPSynThrdNum--;
11 pthread_mutex_unlock(&TCPSynScanlocker);
12 }

```

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

```

1 void* Thread_TCPSynScan(void* param) {
2     /*变量定义*/
3     //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
4     struct TCPSYNThrdParam *p = (struct TCPSYNThrdParam*)param;
5     std::string HostIP = p -> HostIP;
6     unsigned BeginPort = p-> BeginPort;
7     unsigned EndPort = p-> EndPort;
8     unsigned LocalHostIP = p -> LocalHostIP;

```

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。

```

1 TCPSynThrdNum = 0;
2 unsigned LocalPort = 1024;
3 pthread_attr_t attr, lattr;
4 pthread_t listenThreadID, subThreadID;
5 for (unsigned TempPort = BeginPort; TempPort <= EndPort; TempPort++)
6 {
7     //设置子线程参数
8     struct TCPSYNHostThrdParam *pTCPSYNHostParam =
9     new TCPSYNHostThrdParam;
10    pTCPSYNHostParam->HostIP = HostIP;
11    pTCPSYNHostParam->HostPort = TempPort;
12    pTCPSYNHostParam->LocalPort = TempPort + LocalPort;
13    pTCPSYNHostParam->LocalHostIP = LocalHostIP;
14    //将子线程设置为分离状态
15    pthread_attr_init(&attr);
16    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

```

```

17 //创建子线程
18 int ret = pthread_create(&subThreadID, &attr, Thread_TCPSYNHost,
    pTCPSYNHostParam);
19 if (ret== -1)
20 {
21     std::cout << "Can't create the TCP SYN Scan Host thread !" << std::endl;
22 }
23 pthread_attr_destroy(&attr);
24 //子线程数加1
25 pthread_mutex_lock(&TCPSynScanlocker);
26 TCPSynThrdNum++;
27 pthread_mutex_unlock(&TCPSynScanlocker);
28 //子线程数大于100, 休眠
29 while(TCPSynThrdNum > 100) {
30     sleep(3);
31 }
32 }

```

最后, 我们等待子线程数为 0, 返回。

```

1 while(TCPSynThrdNum != 0) {
2     sleep(1);
3 }
4 //返回主流程
5 pthread_exit(NULL);
6 }

```

6. TCP FIN 扫描

这一部分我们需要使用的数据结构如下:

```

1 struct TCPFINHostThrParam
2 {
3     std::string HostIP;
4     unsigned HostPort;
5     unsigned LocalPort;
6     unsigned LocalHostIP;
7 };
8
9 struct TCPFINThrParam
10 {
11     std::string HostIP;
12     unsigned BeginPort;
13     unsigned EndPort;
14     unsigned LocalHostIP;
15 };

```

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

```

1 int TCPFinThrdNum;
2 pthread_mutex_t TCPFinPrintlocker = PTHREAD_MUTEX_INITIALIZER;
3 pthread_mutex_t TCPFinScanlocker = PTHREAD_MUTEX_INITIALIZER;

```

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

```

1 void* Thread_TCPFINHost(void* param) {
2     /*-----与 TCP SYN 扫描类似-----*/
3     //填充 TCP FIN 数据包
4     struct TCPFINHostThrParam *p = (struct TCPFINHostThrParam*)param;
5     std::string HostIP = p -> HostIP;
6     unsigned HostPort = p->HostPort;
7     unsigned LocalPort = p->LocalPort;
8     unsigned LocalHostIP = p->LocalHostIP;
9
10    struct sockaddr_in FINScanHostAddr;
11    memset(&FINScanHostAddr, 0, sizeof(FINScanHostAddr));
12    FINScanHostAddr.sin_family = AF_INET;
13    FINScanHostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
14    FINScanHostAddr.sin_port = htons(HostPort);

```

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

```

1 int FinSock=socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
2 if (FinSock < 0) {
3     pthread_mutex_lock(&TCPFinPrintlocker);
4     std::cout << "Can't creat raw socket !" << std::endl;
5     pthread_mutex_unlock(&TCPFinPrintlocker);
6 }
7
8 int FinRevSock = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
9 if (FinRevSock < 0)
10 {
11     pthread_mutex_lock(&TCPFinPrintlocker);
12     std::cout << "Can't creat raw socket !" << std::endl;
13     pthread_mutex_unlock(&TCPFinPrintlocker);
14 }
15 int flag = 1;
16 if (setsockopt(FinSock, IPPROTO_IP, IP_HDRINCL, (void*)&flag, sizeof(int)) ==
17     -1) {
18     std::cout << "set IP_HDRINCL error.\n";
19 }
20 if (setsockopt(FinRevSock, IPPROTO_IP, IP_HDRINCL, (void*)&flag, sizeof(int)) ==
21     -1) {
22     std::cout << "set IP_HDRINCL error.\n";
23 }

```

填充 TCP 头

```

1 char sendbuf[8192];
2 struct pseudohdr *ptcph = (struct pseudohdr*)sendbuf;
3 struct tcphdr *tcph = (struct tcphdr*)(sendbuf + sizeof(struct pseudohdr));
4
5
6
7 ptcph->saddr = LocalHostIP;
8 ptcph->daddr = inet_addr(&HostIP[0]);
9 ptcph->useless = 0;
10 ptcph->protocol = IPPROTO_TCP;
11 ptcph->length = htons(sizeof(struct tcphdr));

```



```

12
13
14     tcph->th_sport = htons(LocalPort);
15     tcph->th_dport = htons(HostPort);
16     tcph->th_seq = htonl(123456);
17     tcph->th_ack = 0;
18     tcph->th_x2 = 0;
19     tcph->th_off = 5;
20     tcph->th_flags = TH_FIN;
21     tcph->th_win = htons(65535);
22     tcph->th_sum = 0;
23     tcph->th_urp = 0;
24     tcph->th_sum = in_cksum((unsigned short*)ptcph, 20 + 12);

```

封装 IP 头

```

1     IPHeader IPheader(ptcph -> saddr, ptcph -> daddr, IPPROTO_TCP);
2     char temp[sizeof(IPHeader) + sizeof(struct tcphdr)];
3
4     memcpy((void*)temp, (void*)&IPheader, sizeof(IPheader));
5     memcpy((void*)(temp+sizeof(IPheader)), (void*)tcph, sizeof(struct tcphdr));

```

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

```

1     int len = sendto(FinSock, temp, sizeof(IPHeader) + sizeof(struct tcphdr), 0, (
2         struct sockaddr *)&FINScanHostAddr, sizeof(FINScanHostAddr));
3     if(len < 0)
4     {
5         pthread_mutex_lock(&TCPFinPrintlocker);
6         std::cout << "Send TCP FIN Packet error !" << std::endl;
7         pthread_mutex_unlock(&TCPFinPrintlocker);
8     }
9     if(fcntl(FinRevSock, F_SETFL, O_NONBLOCK) == -1)
10    {
11        pthread_mutex_lock(&TCPFinPrintlocker);
12        std::cout << "Set socket in non-blocked model fail !" << std::endl;
13        pthread_mutex_unlock(&TCPFinPrintlocker);
14    }
15    int FromAddrLen = sizeof(struct sockaddr_in);

```

然后开始利用一个循环将收到的数据包存入 buffer 中

```

1     struct timeval TpStart, TpEnd;
2     char recvbuf[8192];
3     struct sockaddr_in FromAddr;
4     std::string SrcIP, DstIP, LocalIP;
5     gettimeofday(&TpStart, NULL); //获得开始接收时刻
6     struct in_addr in_LocalhostIP;
7     do {
8         //调用 recvfrom 函数接收数据包
9         len = recvfrom(FinRevSock, recvbuf, sizeof(recvbuf), 0, (struct sockaddr*)&
            FromAddr, (socklen_t*)&FromAddrLen);

```

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

```

1  if(len > 0)
2  {
3      std::string SrcIP = inet_ntoa(FromAddr.sin_addr);
4      if(1)
5      {
6          //响应数据包的源地址等于目标主机地址
7          struct ip *iph = (struct ip *)recvbuf;
8          int i = iph -> ip_hl * 4;
9          struct tcphdr *tcph = (struct tcphdr *)&recvbuf[i];
10
11          SrcIP = inet_ntoa(iph->ip_src);
12          DstIP = inet_ntoa(iph->ip_dst);
13
14
15          in_LocahostIP.s_addr = LocalHostIP;
16          LocalIP = inet_ntoa(in_LocahostIP);
17
18          unsigned SrcPort = ntohs(tcph->th_sport);
19          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，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

```

1  // std::cout << "-----" << std::endl;
2
3      // std::cout << HostIP << ' ' << SrcIP << std::endl;
4      // std::cout << LocalIP << ' ' << DstIP << std::endl;
5      // std::cout << SrcPort << ' ' << HostPort << std::endl;
6      // std::cout << DstPort << ' ' << LocalPort << std::endl;
7      //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
8      if(HostIP == SrcIP && LocalIP == DstIP && SrcPort == HostPort && DstPort == LocalPort)
9      {

```

然后我们判断数据包类型，给出动作响应，

```

1  if (tcph->th_flags == 0x14)
2  {
3      pthread_mutex_lock(&TCPFinPrintlocker);
4      std::cout << "Host: " << SrcIP << " Port: " << ntohs(tcph -> th_sport) << "
5          closed !" << std::endl;
6      pthread_mutex_unlock(&TCPFinPrintlocker);
7      break;

```

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

```

1  //判断等待响应数据包时间是否超过 3 秒
2  gettimeofday(&TpEnd, NULL);
3  float TimeUse = (1000000 * (TpEnd.tv_sec - TpStart.tv_sec) + (TpEnd.tv_usec -
4      TpStart.tv_usec)) / 1000000.0;
5  if(TimeUse < 5)
6  {
7      continue;

```

```

8   else
9   {
10      //超时, 扫描端口开启
11      pthread_mutex_lock(&TCPFinPrintlocker);
12      std::cout << "Host: " << HostIP << " Port: " << HostPort << " open !" << std::
          endl;
13      pthread_mutex_unlock(&TCPFinPrintlocker);
14      break;
15  }
16  }
17  while(true);

```

最后我们退出子进程，并返回

```

1  delete p;
2  close(FinSock);
3  close(FinRevSock);
4
5  pthread_mutex_lock(&TCPFinScanlocker);
6  TCPFinThrdNum--;
7  pthread_mutex_unlock(&TCPFinScanlocker);
8  }

```

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

```

1  void* Thread_TCPFinScan(void* param) {
2      struct TCPFINThrParam *p = (struct TCPFINThrParam*)param;
3      std::string HostIP = p->HostIP;
4      unsigned BeginPort = p->BeginPort;
5      unsigned EndPort = p->EndPort;
6      unsigned LocalHostIP = p->LocalHostIP;
7
8      TCPFinThrdNum = 0;
9      unsigned LocalPort = 1024;

```

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。

```

1  TCPFinThrdNum = 0;
2  unsigned LocalPort = 1024;
3
4  pthread_attr_t attr, lattr;
5  pthread_t listenThreadID, subThreadID;
6  for (unsigned TempPort = BeginPort; TempPort <= EndPort; TempPort++)
7  {
8      struct TCPFINHostThrParam *pTCPFINHostParam = new TCPFINHostThrParam;
9      pTCPFINHostParam->HostIP = HostIP;
10     pTCPFINHostParam->HostPort = TempPort;
11     pTCPFINHostParam->LocalPort = TempPort + LocalPort;
12     pTCPFINHostParam->LocalHostIP = LocalHostIP;
13
14

```

```

15     pthread_attr_init(&attr);
16     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
17
18
19     int ret = pthread_create(&subThreadID, &attr, Thread_TCPFINHost, pTCPFINHostParam);
20     if (ret == -1)
21     {
22         std::cout << "Can't create the TCP FIN Scan Host thread !" << std::endl;
23     }
24
25     pthread_attr_destroy(&attr);
26     pthread_mutex_lock(&TCPFinScanlocker);
27     TCPFinThrdNum++;
28     pthread_mutex_unlock(&TCPFinScanlocker);
29
30     while (TCPFinThrdNum > 100)
31     {
32         sleep(3);
33     }
34 }

```

最后，我们等待子线程数为 0，返回

```

1     while (TCPFinThrdNum != 0)
2     {
3         sleep(1);
4     }
5
6     std::cout << "TCP FIN scan thread exit !" << std::endl;
7     pthread_exit(NULL);
8 }

```

7. UDP 扫描

这一部分我们使用的数据结构如下:

```

1 struct UDPThrParam
2 {
3     std::string HostIP;
4     unsigned BeginPort;
5     unsigned EndPort;
6     unsigned LocalHostIP;
7 };
8
9 struct UDPScanHostThrParam
10 {
11     std::string HostIP;
12     unsigned HostPort;
13     unsigned LocalPort;
14     unsigned LocalHostIP;
15 };

```

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

```

1 void* UDPScanHost(void* param) {
2     struct UDPScanHostThrParam *p = (struct UDPScanHostThrParam*) param;
3     std::string HostIP = p -> HostIP;
4     unsigned HostPort = p -> HostPort;
5     unsigned LocalPort = p -> LocalPort;
6     unsigned LocalHostIP = p -> LocalHostIP;
7
8     int UDPSock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
9     if(UDPSock < 0)
10    {
11        pthread_mutex_lock(&UDPPrintlocker);
12        std::cout << "Can't creat raw icmp socket !" << std::endl;
13        pthread_mutex_unlock(&UDPPrintlocker);
14    }
15    int on = 1;
16    int ret = setsockopt(UDPSock, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
17
18    if (ret < 0)
19    {
20        pthread_mutex_lock(&UDPPrintlocker);
21        std::cout << "Can't set raw socket !" << std::endl;
22        pthread_mutex_unlock(&UDPPrintlocker);
23    }

```

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

```

1     struct sockaddr_in UDPScanHostAddr;
2     memset(&UDPScanHostAddr, 0, sizeof(UDPScanHostAddr));
3     UDPScanHostAddr.sin_family = AF_INET;
4     UDPScanHostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
5     UDPScanHostAddr.sin_port = htons(HostPort);
6
7     char packet[sizeof(struct iphdr) + sizeof(struct udphdr)];
8     memset(packet, 0x00, sizeof(packet));
9
10    struct iphdr *ip = (struct iphdr *)packet;
11    struct udphdr *udp = (struct udphdr *)(packet + sizeof(struct iphdr));
12    struct pseudohdr *pseudo = (struct pseudohdr *)(packet + sizeof(struct iphdr) -
13        sizeof(struct pseudohdr));
14
15    udp -> source = htons(LocalPort);
16    udp -> dest = htons(HostPort);
17    udp -> len = htons(sizeof(struct udphdr));
18    udp -> check = 0;
19
20    pseudo -> saddr = LocalHostIP;
21    pseudo -> daddr = inet_addr(&HostIP[0]);
22    pseudo -> useless = 0;
23    pseudo -> protocol = IPPROTO_UDP;

```

```

23 pseudo -> length = udp->len;
24
25 udp->check = in_cksum((u_short *)pseudo, sizeof(struct udphdr)+sizeof(struct
    pseudohdr));

```

填充 IP 头

```

1 ip -> ihl = 5;
2 ip -> version = 4;
3 ip -> tos = 0x10;
4 ip -> tot_len = sizeof(packet);
5 ip -> frag_off = 0;
6 ip -> ttl = 69;
7 ip -> protocol = IPPROTO_UDP;
8 ip -> check = 0;
9 ip -> saddr = inet_addr("192.168.1.168");
10 ip -> daddr = inet_addr(&HostIP[0]);

```

然后发送 UDP 数据包，并将 UDPSock 套接字设置为非阻塞模式

```

1 int n = sendto(UDPSock, packet, ip -> tot_len, 0, (struct sockaddr *)&
    UDPScanHostAddr, sizeof(UDPScanHostAddr));
2 if (n < 0)
3 {
4     pthread_mutex_lock(&UDPPrintlocker);
5     std::cout << "Send message to Host Failed !" << std::endl;
6     pthread_mutex_unlock(&UDPPrintlocker);
7 }
8
9 if(fcntl(UDPSock, F_SETFL, O_NONBLOCK) == -1)
10 {
11     pthread_mutex_lock(&UDPPrintlocker);
12     std::cout << "Set socket in non-blocked model fail !" << std::endl;
13     pthread_mutex_unlock(&UDPPrintlocker);
14 }

```

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

```

1 struct timeval TpStart, TpEnd;
2 struct ipicmphdr hdr;
3 gettimeofday(&TpStart, NULL); //get start time
4 do
5 {
6     //receive response message
7     n = read(UDPSock, (struct ipicmphdr *)&hdr, sizeof(hdr));
8
9     if(n > 0)
10     {
11         if((hdr.ip.saddr == inet_addr(&HostIP[0])) && (hdr.icmp.code == 3) && (hdr.
            icmp.type == 3))
12         {
13             pthread_mutex_lock(&UDPPrintlocker);
14             std::cout << "Host: " << HostIP << " Port: " << HostPort << " closed !" <<
                std::endl;
15             pthread_mutex_unlock(&UDPPrintlocker);
16             break;

```

```

17     }
18 }

```

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

```

1  gettimeofday(&TpEnd,NULL);
2  float TimeUse = (1000000 * (TpEnd.tv_sec - TpStart.tv_sec) + (TpEnd.tv_usec -
   TpStart.tv_usec)) / 1000000.0;
3  if(TimeUse < 3)
4  {
5      continue;
6  }
7  else
8  {
9      pthread_mutex_lock(&UDPPrintlocker);
10     std::cout << "Host: " << HostIP << " Port: " << HostPort << " open !" << std
       ::endl;
11     pthread_mutex_unlock(&UDPPrintlocker);
12     break;
13 }
14 }
15 while(true);

```

最后断开链接，线程结束

```

1  //close socket
2  close(UDPSock);
3  delete p;
4  }

```

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

```

1  void* Thread_UDPScan(void* param)
2  {
3
4      // pthread_t subThreadID;
5      // pthread_attr_t attr;
6      // int ret;
7
8      struct UDPThrParam *p = (struct UDPThrParam*) param;
9      std::string HostIP = p -> HostIP;
10     unsigned BeginPort = p -> BeginPort;
11     unsigned EndPort = p -> EndPort;
12     unsigned LocalHostIP = p -> LocalHostIP;

```

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。

```

1  for (unsigned TempPort = BeginPort; TempPort <= EndPort; TempPort++)
2  {
3      UDPScanHostThrParam *pUDPScanHostParam = new UDPScanHostThrParam;
4      pUDPScanHostParam->HostIP = HostIP;
5      pUDPScanHostParam->HostPort = TempPort;
6      pUDPScanHostParam->LocalPort = TempPort + LocalPort;
7      pUDPScanHostParam->LocalHostIP = LocalHostIP;
8      UDPScanHost(pUDPScanHostParam);

```

```

9
10 }
11 //-----exit thread-----
12 std::cout << "UDP Scan thread exit !" << std::endl;
13 pthread_exit(NULL);
14 }

```

8. 端口扫描器程序 Scanner

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，继续执行下面的步骤。

```

1 int main(int argc, char *argv[]) {
2     std::unordered_map<std::string, void(*)(int, char*[])> mapOp = {{ "-h", print_h}, {"-c", print_c}, {"-s", print_s}, {"-u", print_u}, {"-f", print_f}};
3     if (argc != 2) {
4         std::cout << "参数错误, argc = " << argc << std::endl;
5         return -1;
6     }
7     std::string op = argv[1];
8
9
10    if (mapOp.find(op) != mapOp.end()) {
11        mapOp[op](argc, argv);
12        return 0;
13    }
14    return 0;
15 }

```

打印帮助信息函数如下：

```

1 void print_h(int argc, char *argv[]) {
2     std::cout << "Scanner: usage:\n" << "\t" << "[-h] --help information " << std::endl;
3     std::cout << "\t" << "[-c] --TCP connect scan" << std::endl;
4     std::cout << "\t" << "[-s] --TCP syn scan" << std::endl;
5     std::cout << "\t" << "[-f] --TCP fin scan" << std::endl;
6     std::cout << "\t" << "[-u] --UDP scan" << std::endl;
7 }

```

三、 实验结论