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 11 日

目录

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

```
├── include
|   └── header.h
└── src
    ├── TCPConnectScan.cpp
    ├── TCPFINScan.cpp
    ├── TCPSYNScan.cpp
    ├── UDPScan.cpp
    └── 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

```
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;
 };
```

TCP 伪头，用于计算 TCP 头的校验和

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

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));
}
```

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

```
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;
   }
```

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

```
  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));
```

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。

```
//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;
 }
```

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。

```
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;
}
```

4. TCP connect **扫描**

这一部分我们使用的数据结构如下:

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

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

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。

```
int TCPConThrdNum;
pthread_mutex_t TCPConPrintlocker = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t TCPConScanlocker = PTHREAD_MUTEX_INITIALIZER;
```

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。

```
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);

    }
```

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。

```
   //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);
    }
```

然后我们关闭套接字，释放线程锁，线程数量减一。

```
   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
```

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、启始端口、终止端口，然后我们将线程数设置为 0。

```
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
```

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。

```
   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);
        }
    }
```

最后，我们等待子线程数为 0，返回。

```
   while (TCPConThrdNum != 0) {
      sleep(1);
  }
  pthread_exit(NULL);
}
```

5. TCP SYN 扫描

这一部分我们使用的数据结构如下：

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

```

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

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。

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

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

```
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);
```

然后我们创建套接字

```
  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);
```

封装 IP 头

```
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);
 }
```

开始利用一个循环循环接受包到 buffer 中。

```
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);
        }
```

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

```
        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);
```

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

```
            // std::cout << "___________________________" << std::endl;

            // std::cout << HostIP << ' ' << SrcIP << std::endl;
            // std::cout << LocalIP << ' ' << DstIP << std::endl;
            // std::cout << SrcPort << ' ' << HostPort<< std::endl;
            // std::cout << DstPort << ' ' << LocalPort<< std::endl;
            if(HostIP == SrcIP && LocalIP == DstIP && SrcPort == HostPort && DstPort ==
                LocalPort)
            {
```

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。

```

                // 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);
```

最后，我们等待子线程数为 0，返回。

```
 //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

```
void* Thread_TCPSynScan(void* param) {
 /*变量定义*/
 //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
 struct TCPSYNThrParam *p = (struct TCPSYNThrParam*)param;
 std::string HostIP = p -> HostIP;
 unsigned BeginPort = p-> BeginPort;
 unsigned EndPort = p-> EndPort;
 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。

```
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);
        }
    }
```

最后，我们等待子线程数为 0，返回。

```
   while(TCPSynThrdNum != 0) {
      sleep(1);
  }
  //返回主流程
  pthread_exit(NULL);
}
```

6. TCP FIN 扫描

这一部分我们需要使用的数据结构如下：

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

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

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

```
int TCPFinThrdNum;
pthread_mutex_t TCPFinPrintlocker = PTHREAD_MUTEX_INITIALIZER;
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

```
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);
```

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

```
   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";
   }
```

填充 TCP 头

```
   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);
```

封装 IP 头

```
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，并将另一个套接字设置为非阻塞模式进行接收

```
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);
}
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);
```

然后开始利用一个循环将收到的数据包存入 buffer 中

```
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);
```

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

```
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，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

```
// std::cout << "___________________________" << std::endl;

         // std::cout << HostIP << ' ' << SrcIP << std::endl;
         // std::cout << LocalIP << ' ' << DstIP << std::endl;
         // std::cout << SrcPort << ' ' << HostPort<< std::endl;
         // std::cout << DstPort << ' ' << LocalPort<< std::endl;
         //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)
         {
```

然后我们判断数据包类型，给出动作响应，

```
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;
```

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

```
//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);
```

最后我们退出子进程，并返回

```
    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

```
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;
```

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。

```
    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);
        }
    }
```

最后，我们等待子线程数为 0，返回

```
    while (TCPFinThrdNum != 0)
    {
        sleep(1);
    }

    std::cout << "TCP FIN scan thread exit !" << std::endl;
    pthread_exit(NULL);
}
```

7. UDP 扫描

这一部分我们使用的数据结构如下:

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

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

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

```
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);
   }
```

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

```
   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));
```

填充 IP 头

```
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]);
```

然后发送 UDP 数据包，并将 UDPSock 套接字设置为为非阻塞模式

```
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);
}
```

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

```
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;
```

```
        }
    }
```

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

```
    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);
```

最后断开链接，线程结束

```
    //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

```
void* Thread_UDPScan(void* param)
{

    // pthread_t subThreadID;
    // pthread_attr_t attr;
    // int ret;

    struct UDPThrParam *p = (struct UDPThrParam*) param;
    std::string HostIP = p -> HostIP;
    unsigned BeginPort = p -> BeginPort;
    unsigned EndPort = p -> EndPort;
    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。

```
    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);
```

```

    }
    //---------------exit thread----------------------
    std::cout << "UDP Scan thread exit !" << std::endl;
    pthread_exit(NULL);
}
```

8. **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 (mapOp.find(op) != mapOp.end()) {
    mapOp[op](argc, argv);
    return 0;
 }
 return 0;
}
```

打印帮助信息函数如下:

```
void print_h(int argc, char *argv[]) {
    std::cout << "Scaner: usage:\n" << "\t" << "[-h] --help information " << std::endl;
    std::cout << "\t" << "[-c] --TCP connect scan" << std::endl;
    std::cout << "\t" << "[-s] --TCP syn scan" << std::endl;
    std::cout << "\t" << "[-f] --TCP fin scan" << std::endl;
    std::cout << "\t" << "[-u] --UDP scan" << std::endl;
}
```

输入目的 IP 地址和端口信息，并判断是否合法。

```
    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;
    }


}
```

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;
    }
}
```

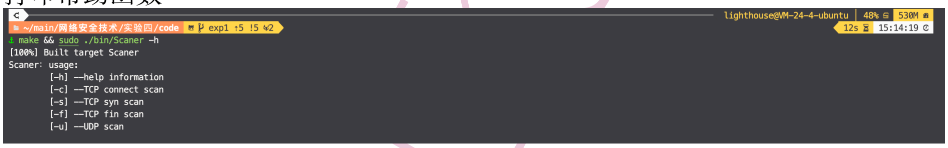
UDP 函数

```
void print_u(int arg, char *argv[]) {
    std::cout << "Begin UDP scan..." << std::endl;
    //create thread for UDP scan
    UDPParam.HostIP = HostIP;
```

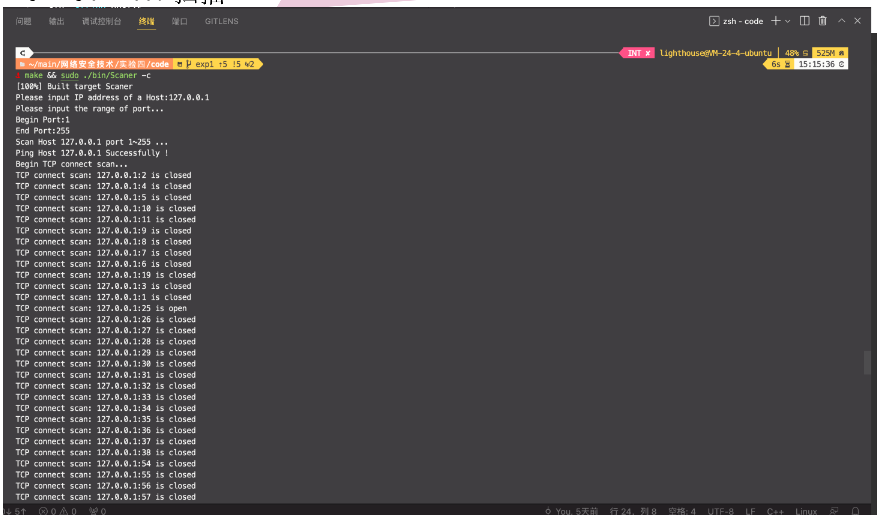
```
    UDPParam.BeginPort = BeginPort;
    UDPParam.EndPort = EndPort;
    UDPParam.LocalHostIP = LocalHostIP;
    ret = pthread_create(&ThreadID, NULL, Thread_UDPScan, &UDPParam);
    if (ret == -1)
    {
        std::cout << "Can't create the UDP scan thread !" << std::endl;
        return;
    }

    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;
    }
}
```

（三） 实验结果

- 打印帮助函数



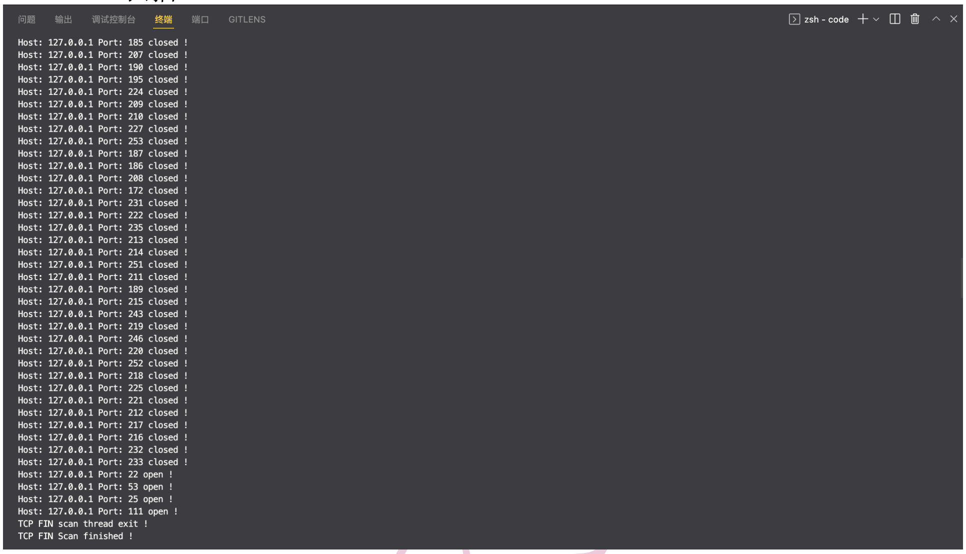
- TCP Connect 扫描



- UDP 扫描

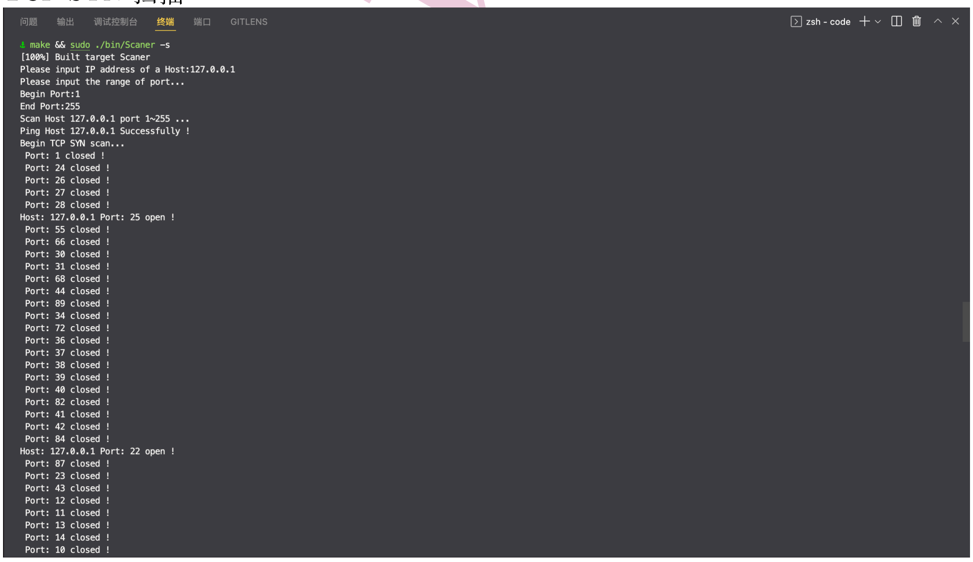
```
~/main/网络安全技术/实验四/code  exp1 ⇡5 !5 ?2                                   10s  15:15:49
make && sudo ./bin/Scaner -u
[100%] Built target Scaner
Please input IP address of a Host:127.0.0.1
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 127.0.0.1 port 1~255 ...
Ping Host 127.0.0.1 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 !
```

- TCP FIN 扫描



```
问题  输出  调试控制台  终端  端口  GITLENS                                      zsh - code

Host: 127.0.0.1 Port: 185 closed !
Host: 127.0.0.1 Port: 207 closed !
Host: 127.0.0.1 Port: 190 closed !
Host: 127.0.0.1 Port: 195 closed !
Host: 127.0.0.1 Port: 224 closed !
Host: 127.0.0.1 Port: 209 closed !
Host: 127.0.0.1 Port: 210 closed !
Host: 127.0.0.1 Port: 227 closed !
Host: 127.0.0.1 Port: 253 closed !
Host: 127.0.0.1 Port: 187 closed !
Host: 127.0.0.1 Port: 186 closed !
Host: 127.0.0.1 Port: 208 closed !
Host: 127.0.0.1 Port: 172 closed !
Host: 127.0.0.1 Port: 231 closed !
Host: 127.0.0.1 Port: 222 closed !
Host: 127.0.0.1 Port: 235 closed !
Host: 127.0.0.1 Port: 213 closed !
Host: 127.0.0.1 Port: 214 closed !
Host: 127.0.0.1 Port: 251 closed !
Host: 127.0.0.1 Port: 211 closed !
Host: 127.0.0.1 Port: 189 closed !
Host: 127.0.0.1 Port: 215 closed !
Host: 127.0.0.1 Port: 243 closed !
Host: 127.0.0.1 Port: 219 closed !
Host: 127.0.0.1 Port: 246 closed !
Host: 127.0.0.1 Port: 220 closed !
Host: 127.0.0.1 Port: 252 closed !
Host: 127.0.0.1 Port: 218 closed !
Host: 127.0.0.1 Port: 225 closed !
Host: 127.0.0.1 Port: 221 closed !
Host: 127.0.0.1 Port: 212 closed !
Host: 127.0.0.1 Port: 217 closed !
Host: 127.0.0.1 Port: 216 closed !
Host: 127.0.0.1 Port: 232 closed !
Host: 127.0.0.1 Port: 233 closed !
Host: 127.0.0.1 Port: 22 open !
Host: 127.0.0.1 Port: 53 open !
Host: 127.0.0.1 Port: 25 open !
Host: 127.0.0.1 Port: 111 open !
TCP FIN scan thread exit !
TCP FIN Scan finished !
```

- TCP SYN 扫描



```
问题  输出  调试控制台  终端  端口  GITLENS                                      zsh - code

make && sudo ./bin/Scaner -s
[100%] Built target Scaner
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: 1 closed !
 Port: 24 closed !
 Port: 26 closed !
 Port: 27 closed !
 Port: 28 closed !
Host: 127.0.0.1 Port: 25 open !
 Port: 55 closed !
 Port: 66 closed !
 Port: 30 closed !
 Port: 31 closed !
 Port: 68 closed !
 Port: 44 closed !
 Port: 89 closed !
 Port: 34 closed !
 Port: 72 closed !
 Port: 36 closed !
 Port: 37 closed !
 Port: 38 closed !
 Port: 39 closed !
 Port: 40 closed !
 Port: 82 closed !
 Port: 41 closed !
 Port: 42 closed !
 Port: 84 closed !
Host: 127.0.0.1 Port: 22 open !
 Port: 87 closed !
 Port: 23 closed !
 Port: 43 closed !
 Port: 12 closed !
 Port: 11 closed !
 Port: 13 closed !
 Port: 14 closed !
 Port: 10 closed !
```

三、 实验遇到的问题及解决方法

(一) 内联函数问题

我们在头文件里定义校验和计算函数时，出现了以下报错

最开始以为是环境变量问题，然后换了几台机子都有这个问题，后面上网查找资料知道了，在我们用 makefile 中将许多文件编译成一个可执行文件的时候，如果所有文件都会引用一个头文件中的函数的时候，这时候将这个函数声明为 static inline 内联函数，否则就会报出这种编译错误。

（二） 数据结构定义问题

最初，我们是在 macos 系统上进行实验的，但是无论我们引用什么包，都会出现以下报错

然后我们意识到，可能是没有这个包，然后我们自己定义了这个数据结构

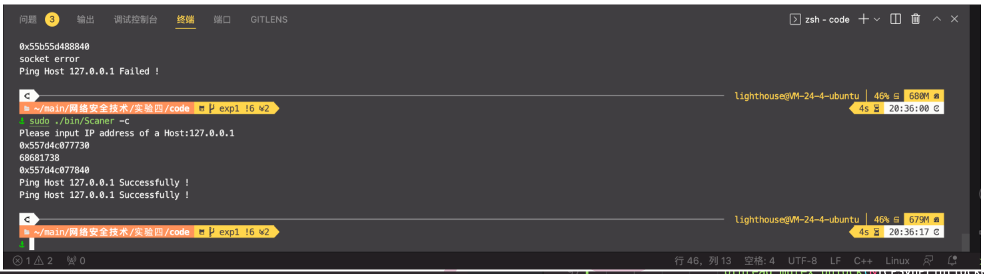
```
struct iphdr {
   unsigned char ihl;
   unsigned char version;
   unsigned char tos;
   unsigned short tot_len;
   unsigned short id;
   unsigned short flag_off;
   unsigned char ttl;
   unsigned char protocol;
   unsigned short check;
   unsigned int saddr;
   unsigned int daddr;
};

struct icmphdr {
   unsigned char type;
   unsigned char code;
   unsigned short check;
   unsigned short identifier;
   unsigned short seq;
   unsigned char data[32];
   union {
      struct {
         unsigned short id;
         unsigned short sequence;
      } echo;
```

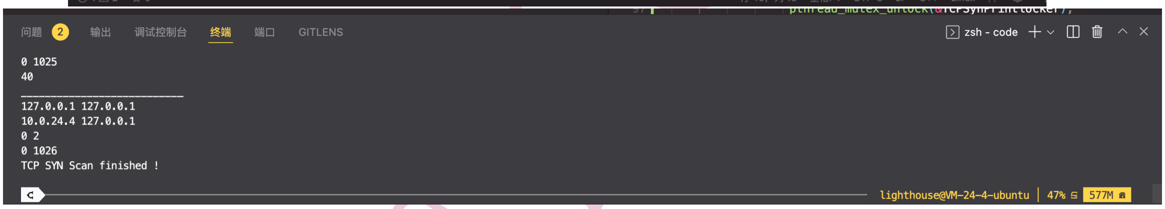
```
      struct {
         unsigned short unused;
         unsigned short mtu;
      } frag;
   } un;
};
```

然后后来我们换到 Linux 系统中进行实验，这个数据机构又出现了报错，貌似是因为 Linux 内核中集成了这个数据结构，所以我们的定义相当于重复了，注释掉便解决了问题。

（三） 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，打印情况如下





最开始我们以为是主函数的代码有问题，然后一直在更改主函数的代码。找了好几天之后，意识到可能是数据结构的问题，然后查看我们的 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 位由于疏忽。当我们更正了这个问题后，一切恢复正常。

（四） 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。

四、 实验结论

本次实验中，虽然提供了大量的框架，但是因为对 Linux 下的 socket 编程不是那么熟悉，所以出现了各种各样的问题，修正花费了一定的时间。但总结来说，我学习到了更多网络通信的知识，提升了我的编程能力和工程能力。同时感谢张老师和助教一个学期的付出和指导帮助，通过这门课我开启了一个全新的知识领域，将来可以涉猎更广泛的知识方向，受益匪浅。