南　　开　　大　　学

网 络 安 全 技 术

端口扫描器的设计与实现

学院：网络安全学院

年级：2021 级

班级：信息安全 1 班

学号：2111252

姓名：李佳豪

手机号：13191110713

2024 年 5 月 24 日

目录

一、 实验目的

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

本章编程训练的目的如下

1. 掌握端口扫描器的基本设计方法
2. 理解 ping 程序，TCP connect 扫描，TCP SYN 扫描，TCP FIN 扫描以及 UDP 扫描的工作原理
3. 熟练掌握 Linux 环境下的套接字编程技术
4. 掌握 Linux 环境下多线程编程的基本方法

二、 实验内容

1. 编写端口扫描程序，提供 TCP connect 扫描，TCP SYN 扫描，TCP FIN 扫描以及 UDP 扫描 4 种基本扫描方式。
2. 设计并实现 ping 程序，探测目标主机是否可达。

三、 实验步骤

(一) 基本原理

1. ICMP 和 Ping

扫描发起主机向目标主机发送一个要求回显（type = 8，即为 ICMP_ECHO）的 ICMP 数据包，目标主机在收到请求后，会返回一个回显（type = 0，即为 ICMP_ECHOREPLY）的 ICMP 数据包。扫描发起主机可以通过是否接收到响应的 ICMP 数据包来判断目标主机是否存在。

因此，通过 Ping 探测主机如下情况 ICMP 报文的格式如下:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Code      |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Identifier          |        Sequence Number        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- 主机在线且允许 **ICMP**：返回 ICMP Echo Reply 报文。
- 主机在线但禁止 **ICMP**：没有响应。
- 主机不在线：没有响应。

2. TCP SYN 和 FIN 报文探测

TCP SYN 扫描和 FIN 扫描是通过发送特定的 TCP 标志位报文来探测目标端口的状态。

1. SYN 扫描通过发送 TCP SYN 报文来探测端口，如果目标端口开放，则会返回 SYN-ACK 报文；如果端口关闭，则返回 RST 报文。但是由于其扫描行为较为明显，SYN 扫描容易被**入侵检测系统**发现，也容易被防火墙屏蔽。同时构造原始的 TCP 数据包也需要较高的系统权限（在 Linux 中仅限于 root 账户）。

   - 端口开放：返回 SYN-ACK 报文，扫描器发送 RST 报文终止连接。
   - 端口关闭：返回 RST 报文。
   - 端口被过滤：没有响应或返回 ICMP 不可达报文。

2. FIN 扫描通过发送 TCP FIN 报文来探测端口，如果端口关闭，则会返回 RST 报文；如果端口开放，则没有响应。但是它的应用具有很大的局限性，由于不同系统实现网络协议栈的细节不同，FIN 扫描只能扫描 Linux/UNIX 系统。对于 **Windows** 系统而言，由于无论端口开放与否，都会返回 RST 数据包，因此对端口的状态无法进行判断

   - 端口开放：没有响应。
   - 端口关闭：返回 RST 报文。
   - 端口被过滤：没有响应或返回 ICMP 不可达报文。

3. TCP Connect 探测

TCP Connect 扫描使用完整的 TCP 三次握手过程来探测端口状态。该方法适用于权限较低的用户，因为它利用操作系统的网络功能完成连接。扫描发起主机只需要调用系统 API connect 尝试连接目标主机的指定端口，如果 connect 成功，意味着扫描发起主机与目标主机之间至少经历了一次完整的 TCP 三次握手建立连接过程，被测端口开放；否则，端口关闭

- 端口开放：完成三次握手后，连接建立，扫描器立即关闭连接。
- 端口关闭：收到 RST 报文，连接无法建立。
- 端口被过滤：没有响应或返回 ICMP 不可达报文。

4. UDP 探测

UDP 扫描通过发送 UDP 报文探测目标端口的状态。一般情况下，当向一个关闭的 UDP 端口发送数据时，目标主机会返回一个 ICMP 不可达（ICMP port unreachable）的错误。UDP 扫描就是利用了上述原理，向被扫描端口发送 0 字节的 UDP 数据包，如果收到一个 ICMP 不可达响应，那么就认为端口是关闭的；而对于那些长时间没有响应的端口，则认为是开放的。

此外，UDP 协议和 ICMP 协议是不可靠协议，没有收到响应的情况也可能是由于数据包丢失造成的，因此扫描程序必须对同一端口进行多次尝试后才能得出正确的结论。

- 端口开放：没有响应或收到应用层报文。
- 端口关闭：收到 ICMP 端口不可达报文。
- 端口被过滤：没有响应或收到 ICMP 不可达报文。

(二) 程序概览

最终程序需要实现多线程对特定 ip 以及选定端口利用不同方式探测端口，因此程序主要考需要考虑:

1. 具体功能函数实现，利用 Ping、Udp、Tcp 等方式探测功能的实现
2. 线程控制以及派生
3. 日志信息
4. 命令行解析

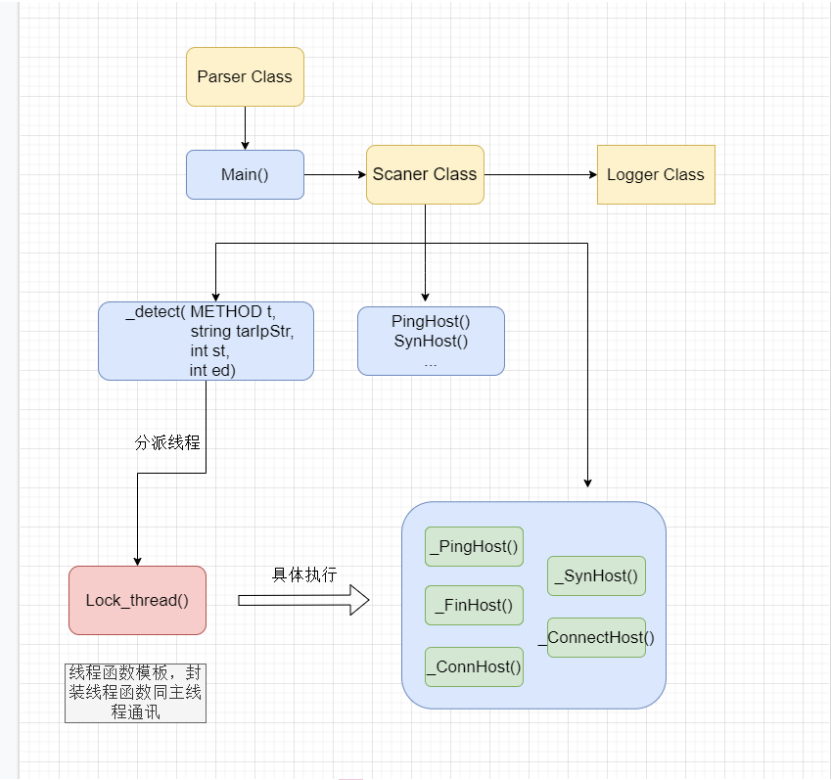

图 1: Enter Caption

程序目录如下:

目录结构

```
Scaner/
    Makefile
    src/
        parse.cpp
        scaner.cpp
        logger.cpp
    include/
        parse.h
        scaner.h
        logger.h
    bin/
```

(三) Scaner：实现探测函数

1. Ping 探测

_PingHost(const std::string tarIpStr, const int p=0) 函数接受一个字符串类型的 IP 地址，并且发送 Ping 报文根据是否有回应判断 ip 是否存在。

1. 初始化并配置 Socket：创建一个 'SOCK_RAW' 的 Socket(协议 'IPPROTO_ICMP')
2. 构造 ICMP 报文：计算 ICMP 报文的大小，并分配相应的内存。初始化 IP 头，设置版本、头长度、总长度、TTL、协议等字段。初始化 ICMP 头，设置类型、代码、标识符、序列号，并计算校验和...
3. 发送 ICMP 报文：设置目标地址结构体。通过 'sendto' 函数发送 ICMP 报文到目标地址。将 Socket 设置为非阻塞模式。
4. 接收并处理响应：
   - 循环接收响应报文。
   - 检查是否接收到来自目标地址的 ICMP 回显应答报文。
   - 如果接收到正确的应答报文，记录日志并返回 1（表示**主机可达**）。
   - 如果在指定时间内未接收到应答报文，则记录日志并返回 0（表示**超时**）。

Ping 探测

```
class Scaner{
public:
    // return 1 reachable 0 timeout
    int _PingHost(const std::string tarIpStr, const int p=0);
};
int Scaner::_PingHost(const std::string tarIpStr, const int p){

    // 变量声明
    // socket setting
    pingSocket = socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);

    // ICMP 报文
    ...
    // ip header init
    ...
    // icmp header init
    struct icmphdr* icmp = (struct icmphdr*)(ip+1);
    icmp->type = ICMP_ECHO;

    // timeval init  ...

    // Calculate ICMP checksum
    icmp->checksum = in_cksum((unsigned short *)icmp, sizeof(struct icmphdr)
        + sizeof(std::tm));

```

```
    // send to tarIp
    ret = sendto(pingSocket,sendBuffer,sendBufSize,0,(struct sockaddr*)&
        taraddr, sizeof(taraddr));
    if(fcntl(pingSocket, F_SETFL, O_NONBLOCK) == -1)

    while(true){
        ret =  recvfrom(pingSocket,recvBuffer,1024,0,NULL,NULL);
        if(ret){
            recvip = (struct ip*)recvBuffer;
            recvicmp = (struct icmp*)(recvBuffer+(recvip->ip_hl*4));
            unsigned int srcIP = (int32_t(recvip->ip_src.s_addr)), dstIP =
                ntohl(int32_t(recvip->ip_dst.s_addr)) ;

            if(srcIP == inet_addr(tarIpStr.c_str()) && dstIP == local_ip
                                && recvicmp->icmp_type == ICMP_ECHOREPLY){
                this->logger.log("ping " + tarIpStr + ": success");
                return 1;
            }

        }
        // 检测timeout
        if (duration.count() >= 3) {
            this->logger.log(tarIpStr + ": waf?");
            return 0 ; //timeout
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

}

};
```

2. Tcp Connect 探测

_ConnectHost(const std::string tarIpStr, const int port)

函数接受一个字符串类型的 IP 地址 `tarIpStr`，以及指定探测的目标端口 `port`，利用 `socket` 库中的 connect 函数对目标 *IP PORT* 发起 TCP 连接，探测端口是否打开。

1. **Socket 初始化**：创建一个 TCP（SOCK_STREAM）Socket。如果 Socket 创建失败，记录错误信息并将返回值设置为-1。
2. **目标地址初始化**：初始化目标地址结构体 `tarAddr`。设置地址族为 `AF_INET`，目标 IP 地址为 `tarIpStr`，目标端口为 `port`。
3. **尝试连接**：
   - 使用 `connect` 函数尝试连接目标主机的指定端口。
   - 如果连接失败（返回值为-1），记录端口关闭的信息，表示该**端口不开放**。

- 如果连接成功，记录端口开放的信息，表示该**端口开放**。

4. **关闭 Socket**：

```
int Scaner::_ConnectHost(const std::string tarIpStr, const int port){

    //std::cout<<"dectect "<<port<<" start"<<std::endl;
    int conSocket = socket(AF_INET,SOCK_STREAM,0), funcret=-1 ,ret;
    sockaddr_in tarAddr;

    if(conSocket == -1){ perror("_ConnectHost()->socket init Error");}

    memset(&tarAddr,0,sizeof(tarAddr));
    tarAddr.sin_family = AF_INET;
    tarAddr.sin_addr.s_addr = inet_addr(tarIpStr.c_str());
    tarAddr.sin_port = htons(port);

    ret =connect(conSocket,(struct sockaddr*)&tarAddr,sizeof(tarAddr));
    if(ret==-1){
        this->logger.log(tarIpStr + ":" + std::to_string(port)+" no");
    }else{
        this->logger.log(tarIpStr+":" + std::to_string(port)+" open");
        funcret = 1;
    }
    close(conSocket);
    return funcret;
}
```

3. TCP FIN、SYN

使用 TCP SYN、FIN 报文探测原理大致相同，因此实现也几乎一致，只是对应 TCP 报头标志位、以及收到目标回复报文的处理部分不同

1. Socket 初始化：创建一个 RAW Socket，协议为 `IPPROTO_TCP`。如果 Socket 创建失败，记录错误信息。

2. 目标地址初始化：初始化目标地址结构体 `tarAddr`。设置地址族为 `AF_INET`，目标 IP 地址为 `tarIpStr`，目标端口为 `port`。

3. 构造 SYN/FIN 报文：分配并初始化发送缓冲区，包含伪头部和 TCP 头部、初始化伪头部，设置源地址、目标地址、协议和 TCP 头部长度，最后初始化 TCP 头部，设置源端口、目标端口、序列号、标志位（SYN）、窗口大小等字段，并计算校验和。

4. 发送 SYN/FIN 报文

5. 循环接收响应报文检查是否接收到来自目标地址的 TCP 响应报文

    - 如果是 SYN 探测，如果接收到 SYN-ACK 报文，记录端口开放的信息，表示对应**端口开放**，如果收到 RST，则表明**端口关闭**，若超时，则表示**不能确定，可能有 waf**。

- 如果是 Fin 探测，如果是超时未接受到响应，则表示**开放**，若收到 RST 回应，表示**不开放**。

SYN 探测

```
int Scaner::_SynHost(const std::string tarIpStr, const int port){

    int synSocket = socket(AF_INET,SOCK_RAW, IPPROTO_TCP), ret;

    // 伪头部
    ptcph->saddr = htonl(local_ip);
    ...
    //tcp
    tcph->th_sport=htons(local_port);
    ...
    tcph->th_sum=in_cksum((unsigned short*)ptcph, 20+12);

    //sendto
    ret = sendto(synSocket, tcph, 20, 0, (struct sockaddr *)&tarAddr,sizeof(
        tarAddr));
    if(fcntl(synSocket, F_SETFL, O_NONBLOCK) == -1){perror("setsockopt");}

    struct ip* recvip;
    struct tcphdr* recvtcp;
    recvBuffer = (char*)malloc(1024);
    auto start = std::chrono::high_resolution_clock::now();

    while(true){
        ret =  recvfrom(synSocket,recvBuffer,1024,0,NULL,NULL);
        if(ret){

            recvip = (struct ip*)recvBuffer;
            recvtcp = (struct tcphdr*)(recvBuffer+(recvip->ip_hl*4));
            unsigned int srcIP = (int32_t(recvip->ip_src.s_addr)), dstIP =
                ntohl(int32_t(recvip->ip_dst.s_addr)) ;

            if(srcIP == inet_addr(tarIpStr.c_str())
                            && dstIP == local_ip
                            && port == ntohs(recvtcp->th_sport)
                            && local_port == ntohs(recvtcp->th_dport)
                            ) {
                if (recvtcp->th_flags == 0x14){ //SYN|ACK
                    this->logger.log(tarIpStr + ":" + std::to_string(port) +
                        " open");
                    return 1;
                }else if(recvtcp->th_flags == 0x12){ //RST
                    this->logger.log(tarIpStr + ":" + std::to_string(port) +
                        " No");
                    return 2;
```

```
                }
            }
        }
        auto duration = std::chrono::duration_cast<std::chrono::seconds>(now
            - start);
        if (duration.count() >= 2) {
            this->logger.log(tarIpStr+":"+std::to_string(port)+" waf?");
            return 0 ; //timeout
        }
    }

}
```

4. UDP 探测

1. Socket 初始化：创建一个 RAW Socket，协议为 `IPPROTO_ICMP`。置 Socket 选项，使手动包含 IP 头。
2.
3. 初始化目标地址结构体 `tarAddr`。设置地址族为 `AF_INET`，目标 IP 地址为 `tarIpStr`，目标端口为 `port`。
4. 构造 UDP 报文：

   分配并初始化发送缓冲区，初始化伪头部，设置源地址、目标地址、协议和 UDP 头部长度; 初始化 UDP 头部，设置源端口、目标端口、长度和校验和; 初始化 IP 头部，设置版本、头长度、总长度、TTL、协议等字段。
5. 发送 UDP 报文：使用 `sendto` 函数发送 UDP 报文到目标地址，并且将 Socket 设置为非阻塞模式; 之后**接收并处理响应**:
   - 如果接收到 ICMP 目标不可达（端口不可达）报文，记录**端口关闭**的信息
   - 如果在指定时间内未接收到响应报文，则认为端口**可能开放**，记录相关信息

UDP 探测

```
int Scaner::_UdpHost(const std::string tarIpStr, const int port){

    // socket
    int udpSocket=socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);
    ret = setsockopt(udpSocket,IPPROTO_IP,IP_HDRINCL,&on,sizeof(on));

    // tar addr_in
    ...
    // udp packetage
    ...
    // calc checksum & 伪头部
    ...
```

```
13      udp->check = in_cksum((u_short *)ptcph,sizeof(struct udphdr)+sizeof(
            struct pseudo_header));
14
15      // ip header
16      ...
17      ip->daddr = inet_addr(tarIpStr.c_str());
18
19      // sendto
20      for(int t=0;t<MAXTRIES;t++){
21          ret = sendto(udpSocket, sendBuffer, ip->tot_len, 0,(struct sockaddr
                *)&tarAddr, sizeof(tarAddr));
22          if(fcntl(udpSocket,F_SETFL, O_NONBLOCK) == -1){perror("setsockopt");}
23
24          auto start = std::chrono::high_resolution_clock::now();
25          while(true){
26              ret =  recvfrom(udpSocket,recvBuffer,1024,0,NULL,NULL);
27              if(ret){
28                  recvip = (struct ip*)recvBuffer;
29                  recvicmp = (struct icmp*)(recvBuffer+(recvip->ip_hl*4));
30                  unsigned int srcIP = (int32_t(recvip->ip_src.s_addr)), dstIP
                        = ntohl(int32_t(recvip->ip_dst.s_addr)) ;
31
32                  if(srcIP == inet_addr(tarIpStr.c_str()) && dstIP == local_ip
33                                  && recvicmp->icmp_type ==
                                        ICMP_DEST_UNREACH
34                                  && recvicmp->icmp_code ==
                                        ICMP_PORT_UNREACH){
35                      this->logger.log(tarIpStr + ":" + std::to_string(port) +
                            " No");
36                      return 2;
37                  }
38              }
39              if (duration.count() >= 3) { //未响应 则开放
40                  break;}
41          }
42      }
43      this->logger.log(tarIpStr + ":"+std::to_string(port)+" open(may be lost)
            ");
44      return 0 ;
45  }
```

（四） 线程与控制

在对一个主机进行端口扫描时，使用多线程方式，此时需要控制线程数量、线程需要向主线程通讯返回结果。

在我的设计中，使用一个函数工厂 *_detect*，负责将具体功能函数放入线程，并且实现线程控制;

> 函数中第一个参数 *METHOD t*，对应具体的功能函数哪个，最终这个 *t* 将**索引一个具体功能函数地址**，通过这个地址和参数在 *lock_thread()* 中进行**动态调用**
>
> 之后则是通过 c++11 提供的线程函数 *thread* 产生线程，mtx 异步锁和 active_threads 则是线程数量控制的实现

1. switch 部分实现 method 变量到目标功能函数索引
2. for 循环部分，针对每一个遍历到的 port，动态调用线程函数执行
3. *lock_thread()* 在线程任务退出时，同主函数通讯，修改相关变量（这里只有关系线程数量的通讯）

函数工厂 _detect

```
1
2  void lock_thread(Scaner *scanner, const std::string tarIpStr, int port, int
       (Scaner::*selectFunction)(const std::string, const int), std::mutex &mtx,
        int &active_threads){
3      (scanner->*selectFunction)(tarIpStr, port);
4      std::lock_guard<std::mutex> lock(mtx);
5      --active_threads;
6  }
7
8  bool Scaner::_detect(METHOD t,const std::string tarIpStr, const int st,const
       int ed){
9
10     std::stringstream ss;
11     ss << "target'addr: "<<tarIpStr<<" detect_scope: "<<st<<"--"<<ed << std
           ::endl;
12     ss << "method: ";
13     int (Scaner::*method)(const std::string, const int) = nullptr;
14
15     switch (t){
16         case PING:
17             ss << "PING"; method = &Scaner::_PingHost; break;
18         case CONNECT:
19             ss << "CONNECT"; method = &Scaner::_ConnectHost; break;
20         case SYN:
21             ss << "SYN"; method = &Scaner::_SynHost; break;
22         case FIN:
23             ss << "FIN"; method = &Scaner::_FinHost; break;
24         case UDP:
```

```
25              ss << "UDP"; method = &Scaner::_UdpHost; break;
26          default:
27              break;
28      }
29      this->logger.log(ss.str());
30
31      // threads crew
32      std::mutex mtx;
33      int active_threads = 0;
34      const int max_threads = this->max_threads;
35
36      for (int port = st; port <= ed; ++port) {
37          {
38              std::unique_lock<std::mutex> lock(mtx);
39              while (active_threads >= max_threads) {
40                  // busy-wait loop
41                  lock.unlock();
42                  std::this_thread::sleep_for(std::chrono::milliseconds(100));
43                  lock.lock();
44              }
45              ++active_threads;
46          }
47
48          std::thread(lock_thread, this, tarIpStr, port, method, std::ref(mtx),
                std::ref(active_threads)).detach();
49      }
50
51      while (true) {
52          {
53              std::lock_guard<std::mutex> lock(mtx);
54              if (active_threads == 0)
55                  break;
56          }
57          std::this_thread::sleep_for(std::chrono::milliseconds(100));
58      }
59
60      this->logger.log("Dectect finished\n");
61      return true;
62  }
```

（五） Logger：单例控制输出

为了防止多线程在终端输出导致的消息错乱，我希望任何时刻只有一个线程拥有输出的权力，因此需要对“输出操作”加锁，若进一步要将输出内容输出到文件，同样还得对“输出文件操作”加锁。将上述功能整合成 **Logger 类**

在 *Logger::log()* 函数内，便实现了对应功能，申请锁，进行“输出到控制台”、“输出到文件”的操作，并且值得注意的是，将 Logger 实现为单例模式，调用 *logger::getInstance()* 保证仅一个 Logger 变量、仅有一个人去“输出”。

Logger 类

```
class Logger {
public:
    // 单例模式
    static Logger& getInstance();
    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;

    void setLogFile(const std::string& filePath);
    void log(const std::string& message){
        std::lock_guard<std::mutex> lock(mutex_);
        std::cout <<message<<std::endl;
        if(is_file){

            if (logFile_.is_open()) {
                logFile_ << message << std::endl;
            } else {
                std::cerr << "Log file is not open." << std::endl;
            }
        }

    };

private:
    Logger() = default;
    bool is_file=false;
    std::ofstream logFile_;
    std::mutex mutex_;
};
```

(六) Parse 命令解析

支持的指令有

short	long	参数	描述
-i	-	无	启用 ICMP 探测
-o	–output	必需	指定输出文件
-s	–scope	必需	指定扫描范围 like:1011-2022, 2912
-m	–method	必需	指定扫描方法

表 1: 支持的指令列表

命令解析

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        print_usage();
        return 1;
    }
    bool Isicmp = false;
    target_url = argv[1];
    int opt,long_index = 0;
    while ((opt = getopt_long(argc, argv, "io:s:m:", long_options, &
        long_index)) != -1) {
        switch (opt) {
            case 'i':
                Isicmp = true;
                break;
            case 'o':
                output_file = optarg;
                break;
            case 's':
                scope = optarg;
                break;
            case 'm':
                method = optarg;
                break;
            default:
                print_usage();
                return 1;
        }
    }

    Scaner scaner("192.168.137.132",1918, 10, output_file);
    if(Isicmp){
        scaner._PingHost(target_url);
        return 0;
    }
    // 验证所有必需的参数是否已提供
```

```
    ...
    // 验证 scope 格式是否有效
    ...
    // 得到扫描范围
    int st, ed;
    split_scope(scope, st,ed);
    // 开始扫描
    if(method == "udp"){
        scaner.UdpHost(target_url,st, ed);
    }else if(method=="syn"){
        scaner.SynHost(target_url,st, ed);
    }else if(method=="fin"){
        scaner.FinHost(target_url,st, ed);
    }else if(method == "conn"){
        scaner.ConnectHost(target_url,st, ed);
    }else{
        std::cerr << "Invalid scan method: " << method << std::endl;
        std::cout << "-m --method: udp syn fin conn"<<std::endl;
    }
    return 0;
}
```

四、 实验结果

(一) Logger 模块测试

Logger test

```
void log_test(Logger& logger, const std::string s) {
    for(int i=0;i<10;i++){
        logger.log(s+":"+ std::to_string(i) + "logging a message");
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        logger.log(s+":"+ std::to_string(i) + "logging a message finished.");
    }
}
int main(){
    Logger& logger1 = Logger::getInstance();
    Logger& logger2 = Logger::getInstance();
    Logger::getInstance().setLogFile("log.txt");

    std::thread t1(log_test, std::ref(logger1),"Alice");
    std::thread t2(log_test, std::ref(logger2),"Bob");
    t1.join();t2.join();
}
```

消息没有出乱，每个线程输出成一行

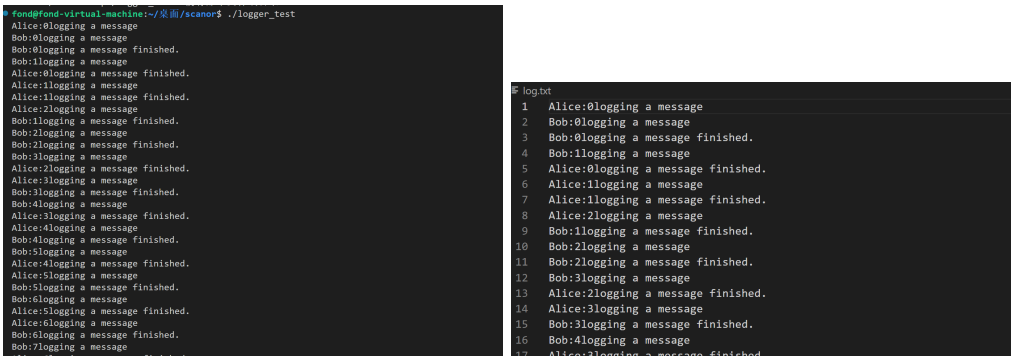

图 2: Logger test

(二) Scaner 模块测试

```
void test(){

std::cout<<"Local ip: 192.168.137.132 Local port: 1918"<<std::endl;
Scaner sc("192.168.137.132");

std::cout<<"———— Test Ping ————"<<std::endl;
std::cout<<"./scaner 192.168.137.1"<<std::endl;
sc._PingHost("192.168.137.1");
std::cout<<"./scaner 192.168.127.123"<<std::endl;
sc._PingHost("192.168.127.123");

std::cout<<std::endl;

std::cout<<"———— Test SYN ————"<<std::endl;
std::cout<<"./scaner 192.168.137.1 -s 8080 -m syn"<<std::endl;
sc._SynHost("192.168.137.1", 8080);
std::cout<<"./scaner 192.168.117.645 -s 7980 -m syn"<<std::endl;
sc.SynHost("192.168.117.645", 7980, 7980);
std::cout<<"./scaner 192.168.137.1 -s 8076-8082 -m syn"<<std::endl;
sc.SynHost("192.168.137.1", 8076, 8080);

std::cout<<std::endl;

std::cout<<"———— Test FIN ————"<<std::endl;
std::cout<<"./scaner 192.168.137.1 -s 8080 -m fin"<<std::endl;
sc._FinHost("192.168.137.1", 8080);
std::cout<<"./scaner 192.168.117.645 -s 7980 -m fin"<<std::endl;
sc.FinHost("192.168.117.645", 7980, 7980);
std::cout<<"./scaner 192.168.137.1 -s 8076-8080 -m fin"<<std::endl;
sc.FinHost("192.168.137.1", 8076, 8080);

std::cout<<std::endl;

std::cout<<"———— Test UDP ————"<<std::endl;
```

```
35      std::cout<<"./scaner 192.168.137.1 -s 8080 -m udp"<<std::endl;
36      sc._UdpHost("192.168.137.1", 8080);
37      std::cout<<"./scaner 192.168.117.645 -s 7980 -m udp"<<std::endl;
38      sc.UdpHost("192.168.117.645", 7980, 7980);
39      std::cout<<"./scaner 192.168.137.1 -s 8076-8080 -m udp"<<std::endl;
40      sc.UdpHost("192.168.137.1", 8076, 8080);
```

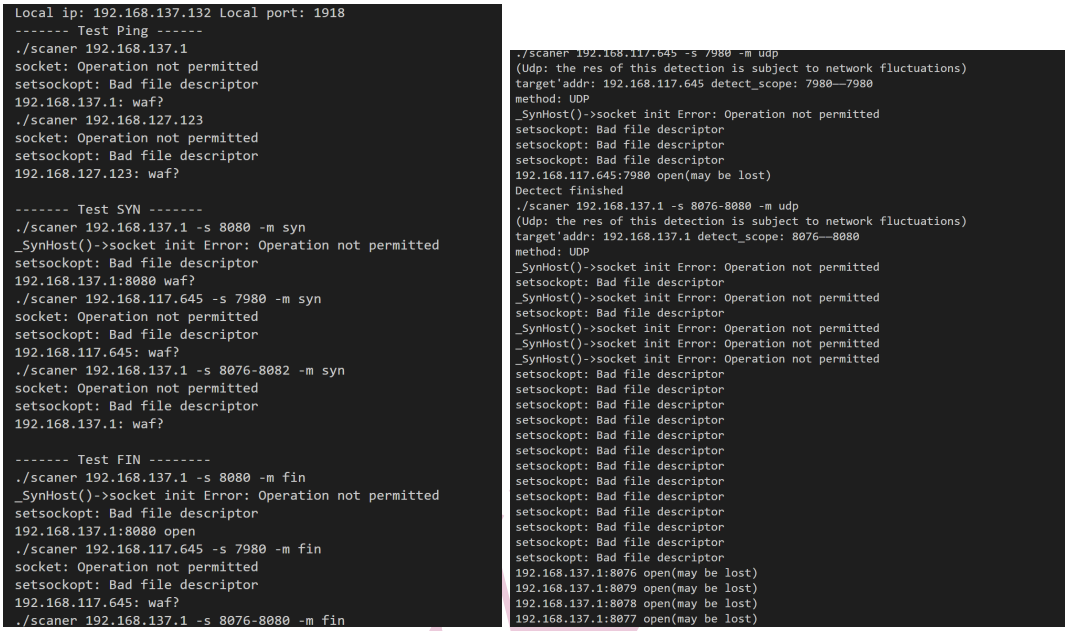

图 3: Scaner test

(三) Parser 整体测试

1. Ping 测试

-i 测试主机存在

./bin/Scaner [ip] -i

测试 192.168.137.1 (win 虚拟机宿主机): 成功
测试 114.xx.xxx.31(linux 阿里云) 成功
测试 114.xx.xxx.21(未知) 成功
测试 114.15.118.21 失败

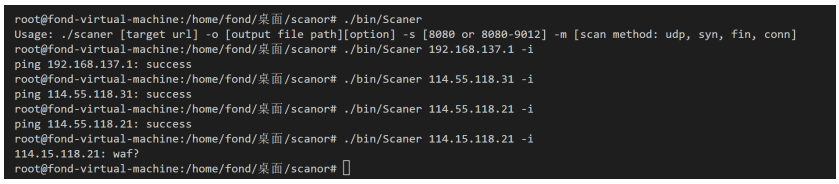

图 4

2. SYN FIN 测试

linux 114.55.xxx.31 已知 80 端口是开放的

```
./bin/Scaner 114.55.xxx.31 -s 80-91 -m syn
./bin/Scaner 114.55.xxx.31 -s 80-91 -m FIN
```

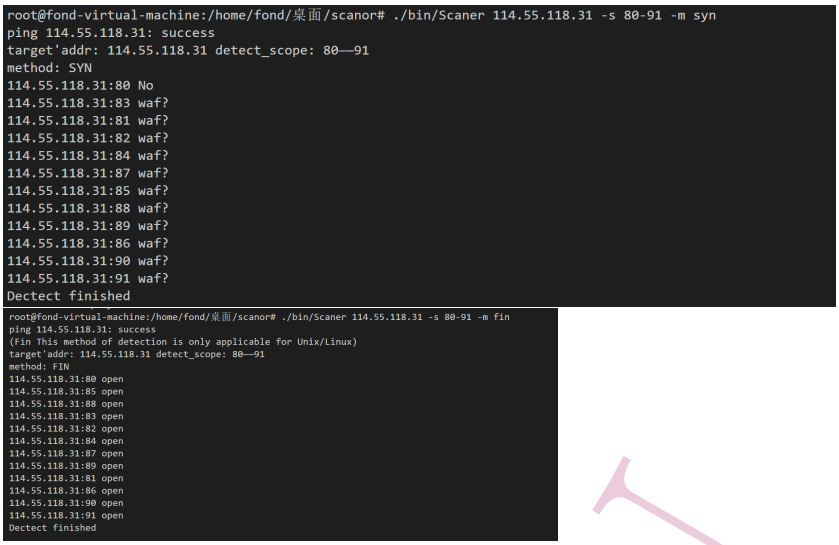

图 5: SYN（左）Fin（右）

结果和预期有些差距，比如 SYN 包检测 80 端口，得到结果是 No，其他任何端口均超时，可能和云服务器的防护机制有关。不过程序在 SYN 探测在 Windows 下是正确的（FIN 无法检测 windows）

3. Conn 测试

```
./bin/Scaner 114.55.xxx.31 -s 20-29 -m conn
./bin/Scaner 192.168.137.1 -s 80-91 -m conn
```

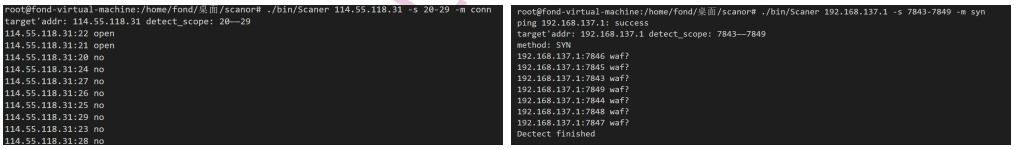

图 6: Linux(左）Win(右)

Win 由于防火墙原因均为探测，linux 则探测到 22(ssh port) 和 21 是开放的。

4. UDP 测试

```
./bin/Scaner 114.55.xxx.31 -s 20-44 -m udp
```

```
root@fond-virtual-machine:/home/fond/桌面/scanor# ./bin/Scaner 114.55.118.31 -s 20-44 -m udp
(Udp: the res of this detection is subject to network fluctuations)
target'addr: 114.55.118.31 detect_scope: 20——44
method: UDP
114.55.118.31:20 open(may be lost)
114.55.118.31:24 open(may be lost)
114.55.118.31:21 open(may be lost)
114.55.118.31:25 open(may be lost)
114.55.118.31:26 open(may be lost)
114.55.118.31:23 open(may be lost)
114.55.118.31:29 open(may be lost)
114.55.118.31:28 open(may be lost)
114.55.118.31:22 open(may be lost)
114.55.118.31:27 open(may be lost)
114.55.118.31:35 open(may be lost)
114.55.118.31:33 open(may be lost)
114.55.118.31:39 open(may be lost)
114.55.118.31:32 open(may be lost)
114.55.118.31:30 open(may be lost)
114.55.118.31:31 open(may be lost)
114.55.118.31:36 open(may be lost)
114.55.118.31:38 open(may be lost)
114.55.118.31:37 open(may be lost)
114.55.118.31:34 open(may be lost)
114.55.118.31:41 open(may be lost)
114.55.118.31:43 open(may be lost)
114.55.118.31:44 open(may be lost)
114.55.118.31:42 open(may be lost)
114.55.118.31:40 open(may be lost)
Dectect finished
```

图 7

五、 实验遇到的问题及其解决方法

(一) 线程使用

问题介绍:

在课程参考实现中，采取了 POSIX 线程（pthread）库中的函数进行相关配置；

由于为了方便调试开始编程时使用的 windows，并不能使用 pthread 库。

解决方案:

使用 C++11 标准库中的 std::thread ，可以在支持 C++11 的所有平台上使用，包括 linux 和 win，且更加简单、异常处理等更加完善。

(二) 多线程打印

问题介绍:

多线程共同向终端打印日志时，可能遇见不同语句重叠到一起的现象，使得难以观看结果

解决方案:

正如上文介绍过的，只需将“打印操作”使用锁机制，因此构造一个 Logger 类，它，使用 mutex 保证每个时刻仅一个线程可以获得打印的权限，同时采取单例模式保证程序执行过程仅一个 logger 对象，因此不会发生多个 logger 共同打印的现象。

(三) 编程 bug

问题介绍：

使用 *int* 开始在记录 ip 这一变量使用了 int，因为编程中的测试均使用的 127.0.0.1，没有什么问题。但使用 192 段的 ip，在 *recv* 抓取包进行 ip 校验无法通过

解决方案：

需要 *unsigned int* 才行，因为 192 开头的 ip 转为 32 位整形最高位为 1。

六、 实验结论

- 在本次实验中，我们使用了 Linux 库中的 Socket，通过具有 root 权限的进程构造和发送 ICMP 请求报文、TCP SYN 报文和 UDP 报文。通过对接收到的响应报文进行分析，我们能够根据一定的规则判断目标 IP 地址是否存在，以及目标端口是否开放。这些网络探测技术包括 ICMP Echo 请求，用于检查主机可达性；TCP SYN 扫描，用于探测 TCP 端口状态；以及 UDP 扫描，用于探测 UDP 端口状态。
- 这些探测方法利用了系统网络协议中规定的面对不同请求的不同回复行为。然而，这些探测方法的效果可能会受到防火墙和其他安全设备的影响，这些设备可能会丢弃、过滤或修改探测报文，从而阻碍我们获取准确的网络状态信息。
- 在实际操作时，由于测试的 windows 主机没有关闭防火墙，基本 tcp、udp 包都无法正常得到回复；而测试的阿里云 linux 主机，也只有 Fin 包可以的得到一些特别的回应，尽管这个回应也不是理想中的。

参考文献

链接
伪头部
icmp 包