



网络空间安全学院  
网络安全技术实验报告

## 实验四：端口扫描器的设计与实现

学院：网安学院  
年级：2021 级  
班级：信息安全一班  
学号：2111408  
姓名：周钰宸  
手机号：13212168838

2024 年 6 月 6 日

## 目录

<b>1 实验目标</b>	<b>3</b>
<b>2 实验原理</b>	<b>3</b>
2.1 Ping 原理 . . . . .	3
2.2 TCP 扫描原理 . . . . .	4
2.2.1 TCP Connect 扫描 . . . . .	4
2.2.2 TCP Syn 扫描 . . . . .	5
2.2.3 TCP Fin 扫描 . . . . .	6
2.3 UDP 扫描原理 . . . . .	7
2.4 Nmap . . . . .	8
2.5 Makefile . . . . .	8
<b>3 实验内容</b>	<b>9</b>
3.1 实验环境 . . . . .	9
3.2 规范习惯 . . . . .	9
3.3 程序交互方式 . . . . .	10
3.3.1 程序运行格式 . . . . .	10
3.3.2 参数功能介绍 . . . . .	11
<b>4 实验步骤</b>	<b>12</b>
4.1 include . . . . .	12
4.1.1 def.hpp . . . . .	12
4.1.2 utils.hpp . . . . .	17
4.1.3 ping.hpp . . . . .	18
4.1.4 TCPConnectScan.hpp . . . . .	20
4.1.5 TCPSynScan.hpp . . . . .	22
4.1.6 TCPFinScan.hpp . . . . .	22
4.1.7 UDPScan.hpp . . . . .	22
4.1.8 Scanner.hpp . . . . .	23
4.2 src . . . . .	24
4.2.1 utils.cpp . . . . .	24
4.2.2 ping.cpp . . . . .	28
4.2.3 TCPConnectScan.cpp . . . . .	36
4.2.4 TCPSynScan.cpp . . . . .	44
4.2.5 TCPFinScan.cpp . . . . .	47
4.2.6 UDPScan.cpp . . . . .	47
4.2.7 Scanner.cpp . . . . .	51
4.3 Makefile . . . . .	55

<b>5 实验结果</b>	<b>56</b>
5.1 展示前准备 . . . . .	56
5.1.1 源主机状态检查 . . . . .	56
5.1.2 目标主机状态检查 . . . . .	56
5.2 实验结果展示 . . . . .	57
5.2.1 显示帮助信息 (-h) . . . . .	57
5.2.2 ping 连通性测试 (-p) . . . . .	57
5.2.3 TCP Connect 扫描 (-c) . . . . .	60
5.2.4 TCP Syn 扫描 (-s) . . . . .	63
5.2.5 TCP Fin 扫描 (-f) . . . . .	66
5.2.6 UDP 扫描 (-u) . . . . .	69
5.2.7 特殊结果分析 . . . . .	71
<b>6 实验遇到的问题及其解决方法</b>	<b>72</b>
6.1 命令行参数校验 . . . . .	72
6.2 多线程扫描 . . . . .	73
6.2.1 资源共享 . . . . .	73
6.2.2 日志线程 . . . . .	74
6.3 管理权限 . . . . .	76
6.4 资源释放 . . . . .	76
<b>7 实验改善方向</b>	<b>77</b>
7.1 其它扫描方式 . . . . .	77
7.2 深度探索 Nmap 原理 . . . . .	77
<b>8 实验结论</b>	<b>77</b>
<b>致谢</b>	<b>79</b>

# 1 实验目标

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。

## 实验目的

本次实验的目的如下：

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

## 实验要求

最终达到的要求为：

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。

本次实验，我按照上述实验要求，完成了能够实现 ping 测试目标主机可达性，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。并在这个基础上，为了提高程序的可交互性和日志信息的完整性美观性等，我又额外实现了-p 选项即单独的 ping 功能，命令行参数功能的全面完整（次数调整 + 端口给定 + 缺省参数）以及日志线程等功能。

除此之外，在这个过程中，为了探究实现的端口扫描器正确性，我配置了另一台 Linux 虚拟机网络，利用 **WireShark 抓包**，对比探究了 Nmap 的实现原理和扫描结果等，进行了全面综合的分析。

# 2 实验原理

## 2.1 Ping 原理

ping 程序是日常网络管理中经常使用的程序。它用于确定本地主机与网络中其它主机的通信情况。因为只是简单地探测某一 IP 地址所对应的主机是否存在，因此它的原理十分简单。

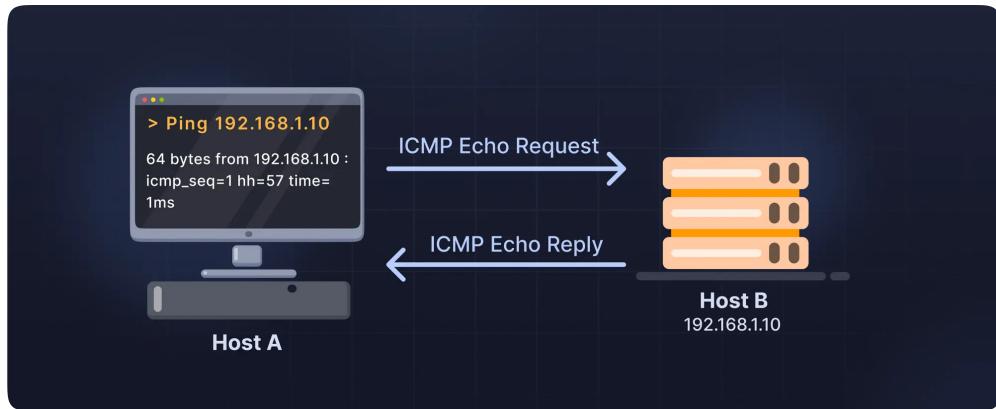


图 2.1: ping 简化原理

如图2.1所示，为简化版的 ping 实现，仅包括发送一个 ICMP 请求数据包和接收一个 ICMP 响应数据包的基本过程。在我们的程序实现中，我们将遵循上述步骤，通过使用 `socket` 套接字和 ICMP 协议，来实现一个简化版的 ping 程序。

## 2.2 TCP 扫描原理

### 2.2.1 TCP Connect 扫描

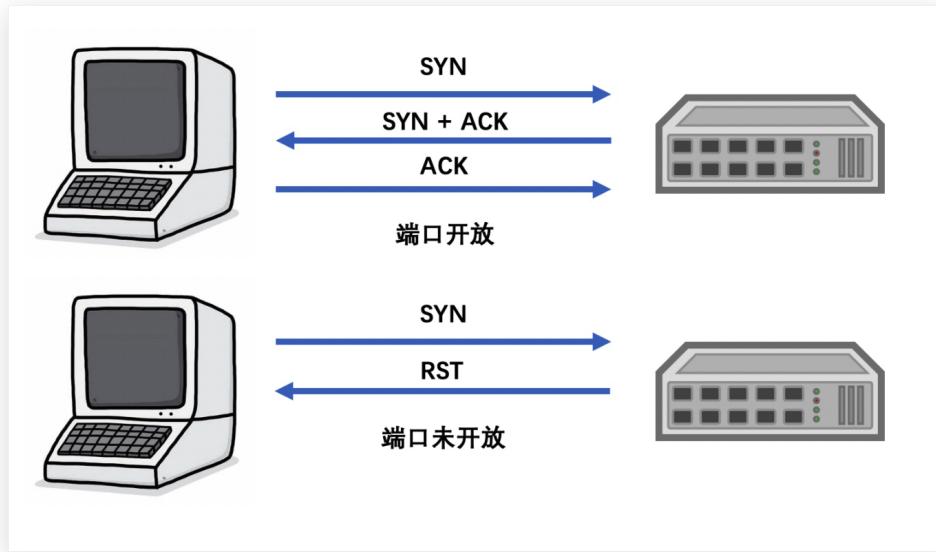


图 2.2: TCP Connect 扫描原理

如图2.2所示，TCP connect 扫描非常简单。扫描发起主机只需要调用系统 API `connect` 尝试连接目标主机的指定端口。如果 `connect` 成功，意味着扫描发起主机与目标主机之间至少经历了一次完整的 TCP 三次握手建立连接过程，被测端口开放；否则，端口关闭。

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

### 2.2.2 TCP Syn 扫描

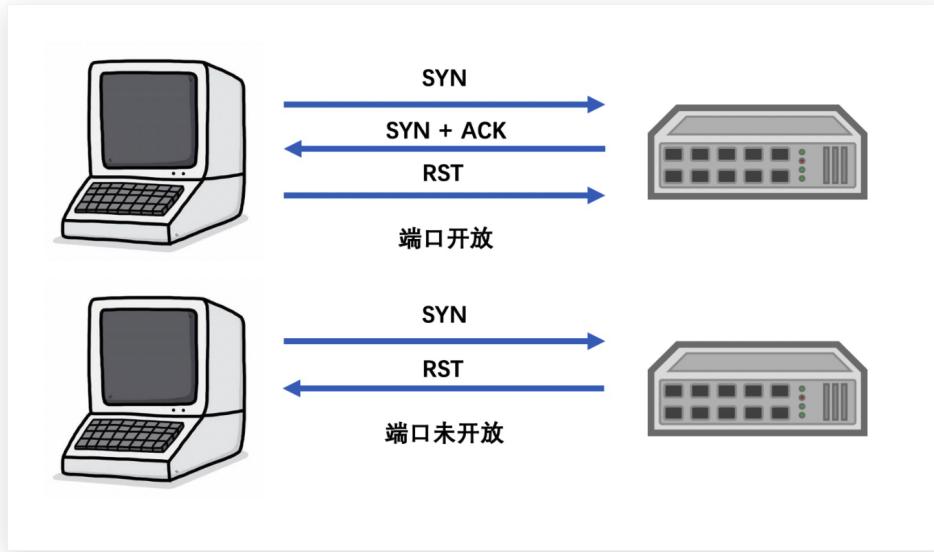


图 2.3: TCP Syn 扫描原理

如图2.3所示，TCP SYN 扫描是使用最广泛的扫描方式，其原理就是向待扫描端口发送 SYN 数据包。如果扫描发起主机能够收到 ACK|SYN 数据包，则表示端口开放；如果收到 RST 数据包，则表示端口关闭。如果未收到任何数据包，且确定目标主机存在，那么发送给被测端口的 SYN 数据包可能被防火墙等安全设备过滤。因为 SYN 扫描不需要完成 TCP 连接的三次握手过程，所以它又被称为半开放扫描。

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

### 2.2.3 TCP Fin 扫描

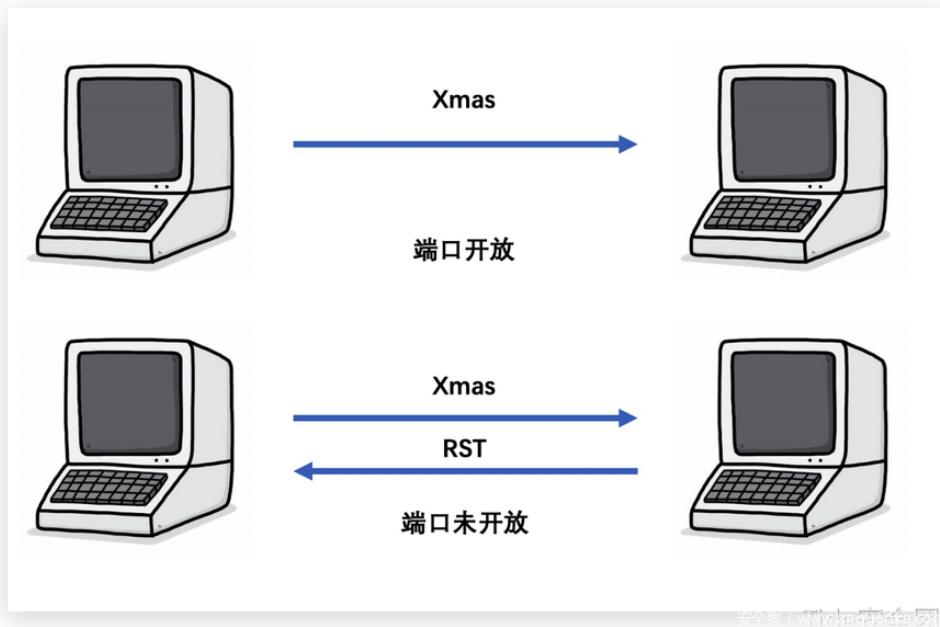


图 2.4: TCP Xmas 扫描原理

如图2.4所示，展示了 TCP 的 Xmas Tree 扫描原理。TCP Fin 扫描相当于是 Xmas Tree 扫描的一部分。Xmas Tree 扫描相当于是将 URG, PSH, FIN 包都设为了 1，而 TCP Fin 扫描是只将 Fin 包设置为 1。

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

FIN 扫描具有良好的隐蔽性，不会留下日志。但是它的应用具有很大的局限性，由于不同系统实现网络协议栈的细节不同，FIN 扫描只能扫描 Linux/UNIX 系统。对于 Windows 系统而言，由于无论端口开放与否，都会返回 RST 数据包，因此对端口的状态无法进行判断。

### 2.3 UDP 扫描原理

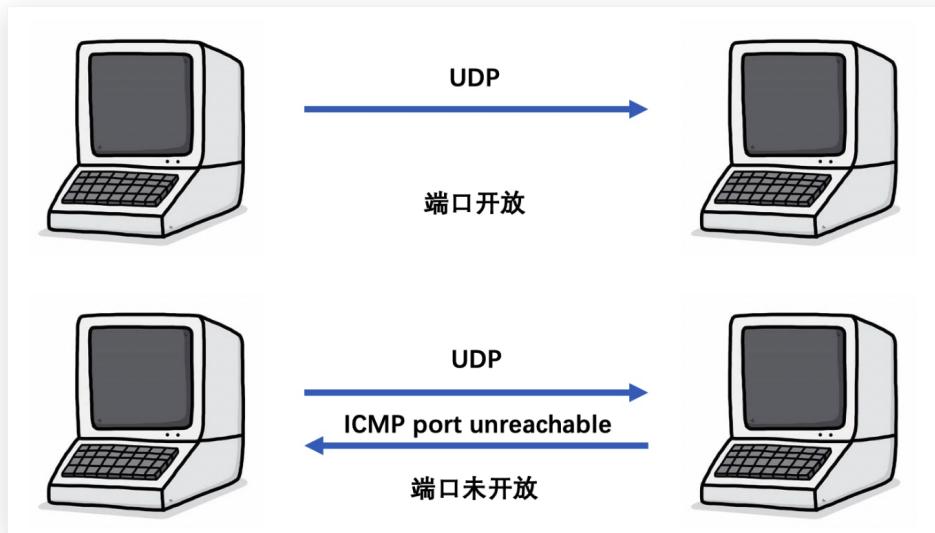


图 2.5: UDP Connect 扫描原理

如图2.5所示，一般情况下，当向一个关闭的 UDP 端口发送数据时，目标主机会返回一个 ICMP 不可达 (ICMP port unreachable) 的错误。UDP 扫描就是利用了上述原理，向被扫描端口发送 0 字节的 UDP 数据包，如果收到一个 ICMP 不可达响应，那么就认为端口是关闭的；而对于那些长时间没有响应的端口，则认为是开放的。

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

## 2.4 Nmap

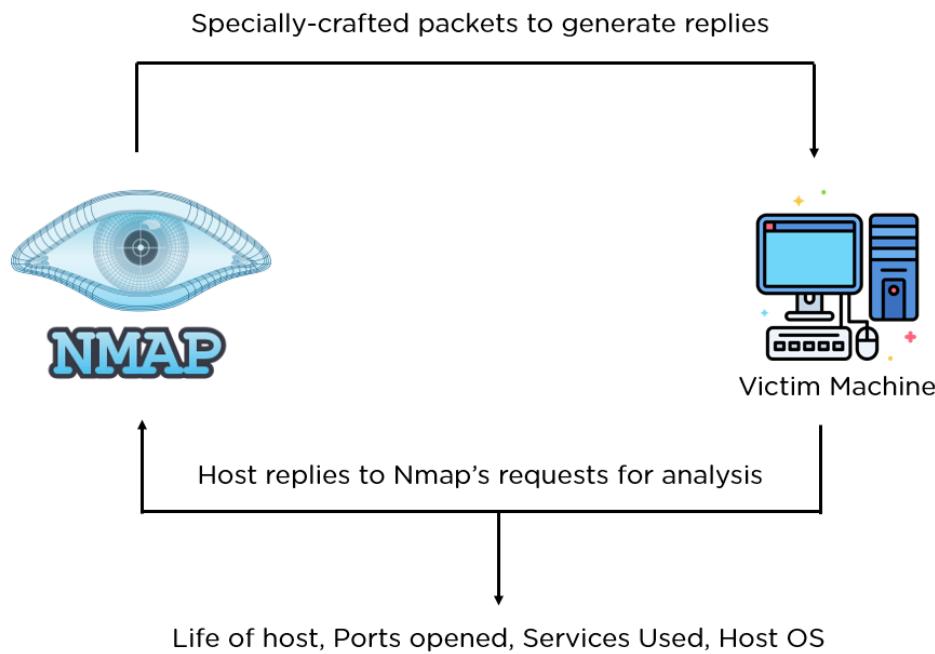


图 2.6: Nmap 原理介绍

如图2.6所示，Nmap（Network Mapper）是一款开源的网络扫描和安全审计工具，被广泛用于网络发现和安全审计。Nmap 使用原始 IP 数据包来确定网络上主机的可达性、所开放的端口、运行的服务（包括操作系统和版本）以及各种网络设备的特征。

Nmap 支持多种扫描技术，包括 **TCP Connect 扫描**、**TCP SYN 扫描**、**UDP 扫描**、**FIN 扫描**、**ACK 扫描**、**Xmas 扫描** 和 **Idle 扫描** 等。通过这些扫描技术，Nmap 可以高效地检测和枚举网络上的活跃主机和开放端口。

Nmap 还支持多种高级功能，如脚本引擎（NSE），用于自动化各种任务，包括网络发现、漏洞检测和安全审计。NSE 脚本可以自定义和扩展，极大地增强了 Nmap 的功能和灵活性。

本次实验为了进一步验证我们实现的端口扫描器的正确性，将会通过 **WireShark 抓包**，将 Nmap 的端口扫描结果和端口扫描器结果进行对比，分析其差异，并探究原因。

## 2.5 Makefile

Linux 环境下，当用户编译文件过多的时候，使用 makefile 可以帮助模块化编译文件，makefile 是一个脚本文件，根据规则，来执行相应的脚本文件，实现自动化编译。

由于本次实验包含了多个 .cpp 实现文件，在编译和链接的时候，逐行地输入重复的命令会显得十分繁琐。稍有疏忽，就会产生错误。为了解决这个问题，本次实验在代码文件的同一目录下创建了一个 makefile 文件，每次编译时，只需在 Shell 命令行中输入 make 命令即可。

### 3 实验内容

本次实验我依据实验目的，在 Linux 环境下编写了一个端口扫描器，利用套接字（socket）正确实现了 ping 程序、TCP connect 扫描、TCP SYN 扫描、TCP FIN 扫描、以及 UDP 扫描。ping 程序在用户输入被扫描主机 IP 地址之后探测该主机是否可达。其它四种扫描在指定被扫描主机 IP，起始端口以及终止端口之后，从起始端口到终止端口对被测主机进行扫描。最后将每一个端口的扫描结果正确地显示出来。具体而言：

#### 实验内容

##### 1. 平台与技术:

- 实验在 Linux 平台下进行。
- 程序正确实现了 ping 程序，与 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. 功能:

- **ping 探测目标主机可达性**: 程序能够指定次数探测任意 IPv4 地址是否可达。
- **TCP 端口扫描**: 程序支持三种 TCP 方式即 TCP Connect, TCP Syn 和 TCP Fin 三种端口扫描方式。
- **UDP 端口扫描**: 程序支持一种 UDP 方式即 UDP 端口扫描方式。
- **额外功能**: -p 选项实现单独根据给定参数对目标主机进行默认次数，指定次数或者一直的 ping 探测可达性。端口扫描范围既可以手动输入，也可以显示给定。

#### 3.1 实验环境

虚拟机软件	VMware Workstation 17 Pro
虚拟机操作系统	Ubuntu 20.04.6 LTS amd64
程序运行平台	x86_64-pc-linux-gnu
实验工具 1	g++ (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0
实验工具 2	CentOS Linux release 7.9.2009 (Core)
实验工具 3	Nmap version 7.80 ( https://nmap.org )

表 1: 本次实验环境及工具

实验平台	IPv4 地址
Source: Ubuntu 20.04.6 LTS amd64	192.168.126.128
Destination: CentOS Linux release 7.9.2009 (Core)	10.136.28.34
Destination: CentOS Linux release 7.9.2009 (Core)	192.168.18.7

表 2: 源与目的 IPv4 地址

#### 3.2 规范习惯

由于本次实验我依旧在 Linux 平台进行编程。并依然严格要求自己，采用了非常规范的工程编程习惯。极为清晰的框架增强了我的代码的可读性和可维护性。这一点也会在具体实验步骤中展开描述。

```

/Lab04
├── bin/ Linux 平台可执行程序
│   └── Scanner 端口扫描器可执行程序
├── include/ C++ 项目头文件
│   ├── def.hpp 宏定义等全局定义
│   ├── utils.hpp 其它功能函数声明
│   ├── ping.hpp ping 头文件
│   ├── TCPConnectScan.hpp TCP Connect Scan 头文件
│   ├── TCPSynScan.hpp TCP Syn Scan 头文件
│   ├── TCPFinScan.hpp TCP Fin Scan 头文件
│   ├── UDPScan.hpp UDP Scan 头文件
│   └── Scaner.hpp 主程序函数声明
├── obj/ C++ 项目目标文件集合, 省略其下具体文件展示
└── src/ C++ 项目实现文件
    ├── utils.cpp 其它功能函数定义实现
    ├── ping.cpp ping 实现文件
    ├── TCPConnectScan.cpp TCP Connect Scan 实现文件
    ├── TCPSynScan.cpp TCP Syn Scan 实现文件
    ├── TCPFinScan.cpp TCP Fin Scan 实现文件
    ├── UDPScan.cpp UDP Scan 实现文件
    └── Scaner.cpp 主程序: 实现 CML 交互以及各自测试扫描等功能
pkgs/ 存放 Wireshark 抓包的各自结果, 辅助分析
├── Nmap/ Nmap 扫描结果集合
│   ├── 10.136.28.34/ 对 Centos 主机即 10.136.28.34 在校园网进行 Nmap 扫描结果集合,
│   │   省略其下具体文件展示
│   └── 192.168.18.7/ 另一组测试结果, 省略其下具体文件展示
└── Port Scanner/ 端口扫描器程序扫描结果集合
    ├── 10.136.28.34/ 对 Centos 主机即 10.136.28.34 在校园网进行端口扫描器扫描结果集
    │   合, 省略其下具体文件展示
    └── 192.168.18.7/ 另一组测试结果, 省略其下具体文件展示
Makefile Makefile 文件辅助完成项目编译链接
README.md 提供对项目的介绍和一些操作的文档提示

```

### 3.3 程序交互方式

#### 3.3.1 程序运行格式

本次的可执行程序在 bin 子目录下的 MD5 文件（没有后缀，方便直接运行）。**运行时的命令参数如下：**

```

sudo ./Scaner [-Option] <target address> {additional parameters}
即
./MD5 [选项] < 目标主机 IPv4 地址 > {附加参数}

```

其中各个参数含义为：

1. **[-Option]**: 选项参数。可选的选项有 `{-h, -c, -s, -f, -u}`。
2. **<target address>**: 目标主机 IPv4 地址。指明用来探测目标主机可达性或者扫描目标主机端口的 IPv4 地址，该参数不可以省略。
3. **additional parameters**: 附加参数。除了必须给定的 `<target address>` 以外还可以提供的参数，具体而言：
  - **目标主机地址可达性探测-p**: 当通过-p 选项进行 ping 探测主机可达性时，可以添加额外参数: `{times}`，即对同一个主机反复探测的次数，具体而言：
    - `{times}=0`: 一直探测直到程序超时或主动结束。
    - `{times}` 缺省：使用默认探测次数即 1 次。
  - **端口扫描 {-c, -s, -f, -u}**: 当进行端口扫描时候，可以添加额外参数: `{begin port}` 和 `{end port}`，即端口扫描的开始端口和终止端口，具体而言：
    - `{begin port}` 和 `{end port}` 显式给定：自动进行扫描，不用交互式给定。
    - `{begin port}` 和 `{end port}` 缺省：程序提示信息让用户自动输入。

上述命令行参数其中 [Option] 和 <target address> 不能省略，其它选项均可以根据需要进行选择，以此实现多功能的交互性需求。

### 3.3.2 参数功能介绍

1. **打印帮助信息**: 在控制台命令行中输入 `sudo ./Scanner -h`，打印程序的帮助信息。帮助信息详细地说明了程序的选项和执行参数。用户可以通过查询帮助信息充分了解程序的功能。
2. **ping 测试目标主机可达性**: 在控制台命令行中输入 `sudo ./Scanner -p < 目标主机 IPv4 地址 > {次数}`，开始 ping 程序探测目标主机可达性。首先检查 IP 地址的正确性，而后若次数没提供，则默认使用 1 次进行探测；若次数为 0，则一直探测直到超时或者手动结束。而后开启扫描进程，将每次探测结果都打印出来。值得注意的是，本次实现仿照了 Linux 的 ping 程序，可以打印传输和接收的比特数，接收到的 ICMP 序列号，TTL 还有单次时间。
3. **TCP Connect 扫描**: 在控制台命令行中输入 `sudo ./Scanner -c < 目标主机 IPv4 地址 > {起始端口} {终止端口}`，开始 TCP connect 扫描。若未提供后两个参数，则程序提示用户输入扫描目标主机的起始端口与终止端口。在检验 IP 地址和端口的正确性之后，程序首先利用 ping 程序探测目标主机的 IP 地址是否可达，如果不可达，则放弃对该主机的扫描；否则开启扫描线程，依次扫描每个端口并将扫描结果实时地显示出来。
4. **TCP Syn 扫描**: 在控制台命令行中输入 `sudo ./Scanner -s < 目标主机 IPv4 地址 > {起始端口} {终止端口}`，开始 TCP syn 扫描。若未提供后两个参数，则程序提示用户输入扫描目标主机的起始端口与终止端口。在检验 IP 地址和端口的正确性之后，程序首先利用 ping 程序探测目标主机的 IP 地址是否可达，如果不可达，则放弃对该主机的扫描；否则开启扫描线程，依次扫描每个端口并将扫描结果实时地显示出来。
5. **TCP Fin 扫描**: 在控制台命令行输入 `sudo ./Scanner -f < 目标主机 IPv4 地址 > {起始端口} {终止端口}`，开始 TCP fin 扫描。若未提供后两个参数，则程序提示用户输入扫描目标主机

的起始端口与终止端口。在检验 IP 地址和端口的正确性之后，程序首先利用 ping 程序探测目标主机的 IP 地址是否可达，如果不可达，则放弃对该主机的扫描；否则开启扫描线程，依次扫描每个端口并将扫描结果实时地显示出来。

6. **UDP 扫描：**在控制台命令行输入 `sudo ./Scanner -u < 目标主机 IPv4 地址 > {起始端口} {终止端口}`，开始 UDP 扫描。若未提供后两个参数，则程序提示用户输入扫描目标主机的起始端口与终止端口。在检验 IP 地址和端口的正确性之后，程序首先利用 ping 程序探测目标主机的 IP 地址是否可达，如果不可达，则放弃对该主机的扫描；否则开启扫描线程，依次扫描每个端口并将扫描结果实时地显示出来。

## 4 实验步骤

接下来我会按照之前所述我的程序代码架构以此介绍我的各部分代码，将会结合具体代码解释其思路。

### 4.1 include

#### 4.1.1 def.hpp

该头文件是其它头文件所包含的底层头文件，主要通过宏定义的方式定义了一系列辅助程序功能的常量和结构体。具体而言：

1. `#define MAX_PORT 65535` 与 `#define MIN_PORT 0`  
定义了扫描端口的范围，即最小端口号和最大端口号。
2. `#define DEFAULT_LOCAL_HOST_IP "127.0.0.1"` 与  
`#define DEFAULT_LOCAL_INET_IP "192.168.126.128"`  
默认本地主机的 IPv4 地址，用于网络通信。
3. `#define DEFAULT_LOCAL_PORT 80`  
默认的本地主机端口号，用于网络通信。
4. `#define DEFAULT_HOST_IP "127.0.0.1"` 与 `#define DEFAULT_HOST_TEST_IP "110.242.68.66"`  
默认被扫描主机的 IPv4 地址，其中 110.242.68.66 是 www.baidu.com 的 IPv4 地址，用于测试。
5. `#define MAX_BUFFERS_SIZE 8192`  
定义了缓冲区的最大大小，用于数据传输。
6. `#define MAX_PING_TIMEOUT 5, #define MAX_SYN_TIMEOUT 3, #define MAX_FIN_TIMEOUT 3, #define MAX_UDP_TIMEOUT 5`  
定义了不同扫描方法的最大超时时间，分别用于 ping 扫描、TCP SYN 扫描、TCP FIN 扫描和 UDP 扫描。由于通常而言,SYN 扫描是最快的，UDP 扫描较慢，因此二者超时时间设置的不同。
7. `#define DEFAULT_PING_TIMES 3`  
默认的 ping 次数，用于探测主机是否可达。

8. `#define ARG_HELP "-h", #define ARG_PING "-p", #define ARG_CONNECT_SCAN "-c", #define ARG_SYN_SCAN "-s", #define ARG_FIN_SCAN "-f", #define ARG_UDP_SCAN "-u"`

定义了命令行参数，用于选择不同的扫描方法。对应于 {-h, -c, -s, -f, -u}。

9. `#define SUCCESS 1, #define FAILURE 0, #define ERROR -1, #define TIMEOUT -2`

定义了程序返回的状态码，分别表示成功、失败、错误和超时。失败指的是端口关闭，错误指的是扫描过程中某些步骤出现了问题，二者性质不同。

10. `#define MANUAL 0 与 #define AUTO 1`

定义了手动和自动输入的模式，分别表示手动输入和自动输入。为了实现更好的命令行交互方式。

此外，该头文件还定义了一些重要的结构体和模板类，例如：

#### 1. ThreadSafeQueue

线程安全的队列模板类，用于多线程通信。包含基本的操作，如 push 和 pop。

---

```
1     /* Class Declarations */
2 // Mutil-Thread Communication Class - ThreadSafeQueue
3 template <typename T>
4 class ThreadSafeQueue
5 {
6 private:
7     /**
8      * Basic variables for the thread-safe queue
9      */
10    std::queue<T> queue;
11    std::mutex mtx;
12    std::condition_variable cv;
13
14 public:
15     /**
16      * Operations for the thread-safe queue
17      */
18     // Push the value into the thread-safe queue
19     void push(T value)
20     {
21         /**
22          * Push the value into the thread-safe queue
23          * @param value: The value to push
24          */
25         std::lock_guard<std::mutex> lock(mtx);
26         queue.push(std::move(value));
27     }
28 }
```

```

27     cv.notify_one();
28 }
29 // Pop the value from the thread-safe queue
30 T pop()
31 {
32     /**
33      * Pop the value from the thread-safe queue
34      * @return: The value popped
35      */
36     std::unique_lock<std::mutex> lock(mtx);
37     cv.wait(lock, [this]
38     {
39         { return !queue.empty(); });
40         T value = std::move(queue.front());
41         queue.pop();
42         return value;
43     });
44     bool empty()
45     {
46         /**
47          * Check if the queue is empty
48          * @return: True if the queue is empty, otherwise False
49          */
50         std::lock_guard<std::mutex> lock(mtx);
51         return queue.empty();
52     };

```

## 2. pseudohdr

伪 TCP/UDP 头部结构体，用于计算 TCP 和 UDP 包的校验和，实际上并不存在，只是用于填充信息后计算一个校验和。

---

```

1      /* Struct Definitions */
2 // Pseudo TCP/UDP Header
3 struct pseudohdr
4 {
5     /**
6      * This struct is used for the pseudo header of the TCP and UDP package
7      * To calculate the checksum of the TCP and UDP package
8      */
9     uint32_t saddr;    // Source IP address
10    uint32_t daddr;   // Destination IP address
11    uint8_t useless; // To fill the 8 bits of 0

```

---

```

12     uint8_t protocol; // Protocol type
13     uint16_t length; // The length of the TCP package
14 };
15

```

---

### 3. ipicmphdr

IP-ICMP 头部结构体，用于接收 UDP 扫描的响应。即由 IP 头和 ICMP 头组成。

---

```

1 // IP-ICMP Header
2 struct ipicmphdr
3 {
4     /**
5      * This struct is used for the IP and ICMP header
6      * To use in receiving the UDP scan response
7      */
8     struct iphdr ipHeader; // The IP header
9     struct icmphdr icmpHeader; // The ICMP header
10 };

```

---

### 4. LogMessage

多线程通信结构体，用于存储日志信息，包括端口号和日志消息。

---

```

1 // Mutil-Thread Communication Struct - LogMessage
2 struct LogMessage
3 {
4     /**
5      * The struct for storing the log message:
6      * (1) The port number to output the log message in order
7      * (2) The log message
8      */
9     int port;
10    std::string message;
11 };

```

---

### 5. TCPConnectHostThreadParam, TCPConnectThreadParam, TCPSynHostThreadParam, TCPSynThreadParam, TCPFinHostThreadParam, TCPFinThreadParam, UDPScanHostThreadParam, UDPScanThreadParam

分别用于存储不同扫描方法的参数，包括主机 IP 地址、端口号、本地主机 IP 地址和端口号等。用于在多线程扫描中作为单实参传递多个参数，带 Host 关键字的是用来扫描具体端口的参数，不带的是扫描端口范围的线程函数所用到的参数。

```
1 // TCP Connect Scan Struct
2 struct TCPConnectHostThreadParam
3 {
4     /**
5      * The struct for the parameters of the TCP Connect Scanning for the
6      * specific port
7      */
8     std::string hostIP; // The IP address of the host
9     int port;           // The port for scanning
10 };
11
12 struct TCPConnectThreadParam
13 {
14     /**
15      * The struct for the parameters of the TCP Connect Scanning for the range
16      * of ports
17      */
18     std::string hostIP; // The IP address of the host
19     int beginPort;    // The begin port for scanning
20     int endPort;       // The end port for scanning
21 };
22
23 // TCP Syn Scan struct
24 struct TCPSynHostThreadParam
25 {
26     /**
27      * The struct for the parameters of the TCP Syn Scanning for the specific
28      * port
29      */
30     std::string hostIP;       // The IP address of the host
31     int port;                 // The port for scanning
32     std::string localHostIP; // The IP address of the local host
33     int localPort;           // The port of the local host
34 };
35
36 struct TCPSynThreadParam
37 {
38     /**
39      * The struct for the parameters of the TCP Syn Scanning for the range of
40      * ports
41      */
42 }
```

```
38     std::string hostIP;           // The IP address of the host
39     std::string localHostIP;      // The IP address of the local host
40     int beginPort;              // The begin port for scanning
41     int endPort;                // The end port for scanning
42     int localPort;              // The port of the local host
43 };
44
45 // TCP Fin Scan struct
46 struct TCPFinHostThreadParam
47 {
48     /**
49      * The struct for the parameters of the TCP Fin Scanning for the specific
50      * → port
51      */
52     std::string hostIP;           // The IP address of the host
53     int port;                   // The port for scanning
54     std::string localHostIP;     // The IP address of the local host
55     int localPort;              // The port of the local host
56 };
57 /**
58  * TCPFinHostThreadParam, TCPFinThreadParam
59  * UDPScanHostThreadParam, UDPScanThreadParam
60  * 上述两者和 TCP Syn 的结构体定义完全一致，在此省略
61  */
```

#### 4.1.2 utils.hpp

该头文件包含了为了能够顺利进行 ping 探测主机可达性以及 TCP/UDP 端口扫描而声明的功能函数。它们主要用于保证上述过程的正确性，完整性，鲁棒性。

##### 1. in\_cksum

主要用于通过伪头部计算 TCP 和 UDP 数据头部的校验和。

##### 2. isValidIPv4

主要用于在 Scanner.cpp 即主函数用户输入 IPv4 地址作为命令行参数时候进行检查，防止输入不正确的 IPv4 地址。保证程序的正确性和鲁棒性。

##### 3. isValidPort

主要用于在 Scanner.cpp 即主函数用户输入端口作为命令行参数时候进行检查，防止输入不正确的端口范围。保证程序的正确性和鲁棒性。

##### 4. logProcessingThread

为了保证多线程端口扫描器进行扫描时能够对控制台资源不进行竞争，保证日志输出的有序和美观。为此实现的安全线程处理函数。

具体函数定义详见 `utils.cpp` 对其实现的介绍。具体代码如下：

---

```

1 // Declarations of global utility functions utilized in other cpp files
2
3 #ifndef UTILS_HPP
4 #define UTILS_HPP
5
6 #include "def.hpp" // Include the constants and macros for the global use
7 #include <cstdint> // uint16_t and other unsigned integer
8 #include <cstring> // memset
9 #include <regex>   // std::regex_match
10
11 uint16_t in_cksum(uint16_t *addr, int len);    // Compute the checksum of headers
12 bool isValidIPv4(const char *pszIPAddr);        // Check if the input is a valid IPv4
   → address
13 bool isValidPort(int beginPort, int endPort); // Check if the begin and end ports
   → are valid for out of range
14
15 /* Mutil-Thread Communication LogThread */
16 /* Global Functions Definitions */
17 void logProcessingThread(ThreadSafeQueue<LogMessage> &logQueue, int beginPort, int
   → endPort); // The log processing thread for the log message
18
19 #endif // UTILS_HPP

```

---

#### 4.1.3 ping.hpp

主要用于定义 ping 程序测试目标主机可达性，即单独的-p 选项以及在端口扫描前自动进行可达性测试即-c, -s, -f, -u 选项时候会用到一些成员变量和函数。由于 ping 程序主要通过 ICMP 协议实现，因此将其封装在了一个类 **ICMPUtil** 中。除了套接字，地址，端口，地址结构体外，其中比较特殊的有：

##### 1. 特殊的成员变量

- **int on**  
主要用于手动设置 IP 头。`setsockopt` 中作为取值为 1 的参数传入。
- **struct timeval tp, waitingStartTP 和 waitingEndTP**  
用于记录超时时间，单词探测时间的 `timeval` 结构体。
- **struct ip, icmp 的 send(recv)IPHeader, send(recv)ICMPHeader**  
这里没有使用 Linux 平台下的 `iphdr` 和 `icmphdr`，这是为了提高程序的泛性和可拓展性，实现不局限于 Linux 运行的跨平台程序。采用的是 `ip` 和 `icmp` 结构体。

## 2. 重要的成员函数

- **ICMPUtil(std::string hostIP, int localPort, std::string localHostIP)**  
带有参数的构造函数，来构造 ping 程序所需的 ICMP 工具变量和函数。
- **INT ping(int times)**  
ping 程序主要功能函数。根据给定的参数 **times** 确定来探测的次数，返回探测是否成功等的结果。

具体函数定义详见 `ping.cpp` 对其实现的介绍。完整代码为：

```
1 // The constants, variables and declarations about the Ping Scanning using ICMP
2 → protocol
3
4 #ifndef PING_HPP
5 #define PING_HPP
6
7 #include "def.hpp"
8 #include "utils.hpp"
9 #include <string>           // std::string
10 #include <cstring>          // memset
11 #include <iostream>          // std::cout
12 #include <sys/time.h>        // gettimeofday
13 #include <sys/socket.h>      // socket
14 #include <netinet/in.h>      // struct sockaddr_in
15 #include <arpa/inet.h>        // inet_pton
16 #include <unistd.h>          // close
17 #include <fcntl.h>            // fcntl
18
19 class ICMPUtil
20 {
21     /**
22      * Declaration of the variables
23      */
24     /* IP address and Ports */
25     // Sending ICMP Package
26     std::string localHostIP; // The IP address of the local host
27     int localPort;           // The port of the local host
28     std::string hostIP;       // The IP address of the host
29
30     // Receiving ICMP Package
31     std::string sourceHostIP; // The IP address of the source host
```

```

32     std::string destHostIP;    // The IP address of the destination host
33
34     // Socket Variables
35     int pingSocket;           // socket for ICMP
36     int on;                  // the flag for the socket
37     int sendBufferSize;       // the size for sending buffer
38     char *sendBuffer;         // the buffer for sending
39     char *recvBuffer;         // the buffer for receiving
40     struct timeval *tp;       // the timeval struct as timestamp
41     struct timeval waitingStartTP; // the timeval struct as the start time for
        ↳ waiting
42     struct timeval waitingEndTP; // the timeval struct as the end time for
        ↳ waiting
43     INT ret;                 // return results to determine the success of
        ↳ the operation
44     struct sockaddr_in pingHostAddr; // the destination address to send ICMP packet
45     struct sockaddr_in fromHostAddr; // the source address of the received packet
46     int addrLen;              // the length of the address
47     int pingtimes;            // the times for sending the ICMP packet
48     bool flag;                // the flag to determine whether it is the
        ↳ initial sending
49
50     // Struct Headers
51     struct ip *sendIPHeader;   // IP Header for sending
52     struct icmp *sendICMPHeader; // ICMP Header for sending
53     struct ip *recvIPHeader;   // IP Header for receiving
54     struct icmp *recvICMPHeader; // ICMP Header for receiving
55
56 public:
57     ICMPUtil();
58     ICMPUtil(std::string hostIP, int localPort, std::string localHostIP);
59     ~ICMPUtil();
60     INT ping(int times);
61 };
62 #endif // PING_HPP

```

#### 4.1.4 TCPConnectScan.hpp

该头文件定义了 TCPConnectScanUtil 工具类, 即单独的-c 选项。其中包含了用于 TCP Connect 扫描的相关成员变量和函数声明。为了能够提高程序的运行效率和性能, 因此采用了多线程的实现方式, 其中包含了辅助器实现的多个变量。为了能够和线程函数配合使用, 大部分变量和函数通过 static 关键字定义为了静态, 其中比较特殊的有:

## 1. 特殊的成员变量

- **static int TCPConnectThreadNum**  
用于实时记录和统计当前线程个数，方便主线程知道何时退出。
- **static ThreadSafeQueue<LogMessage> logQueue**  
为了线程安全实现的日志队列，方便日志信息的完整有序输出。
- **static pthread\_mutex\_t**  
一些 `mutex` 定义的线程锁，用于解决多线程资源竞争问题。避免同时访问等造成读取冲突。

## 2. 重要的成员函数

- **static void \*Thread\_TCPConnectHost(void \*param)**  
静态函数，用于具体对目标主机的某个端口开启扫描，多线程实现，其返回值由于被设计为了线程分离因此不再重要，传入的参数是 `TCPConnectHostThreadParam`。
- **static void \*Thread\_TCPConnectScan(void \*param)**  
静态函数，用于对目标主机的一个范围内的端口开启扫描，多线程实现，创建 `TCPConnectHostThreadParam` 结构体，并传入函数调用 `Thread_TCPConnectHost` 进行具体端口的扫描。返回结果为正确与否。

具体代码为：

---

```

1 // The constants, variables and declarations about the TCP Connect Scanning using
2   → TCP protocol
3
4 #ifndef TCP_CONNECT_SCAN_HPP
5 #define TCP_CONNECT_SCAN_HPP
6
7 #include "def.hpp"
8 #include "utils.hpp"
9 #include <string>           // std::string
10 #include <pthread.h>        // pthread_t, pthread_attr_t, pthread_create,
11   pthread_join, pthread_attr_init, pthread_attr_destroy
12 #include <mutex>            // std::mutex
13 #include <unistd.h>          // sleep
14 #include <iostream>           // std::cout, std::endl
15 #include <sys/socket.h> // socket, AF_INET, SOCK_STREAM
16 #include <arpa/inet.h> // sockaddr_in, inet_addr, htons
17
18 class TCPConnectScanUtil
19 {

```

```

20 private:
21     /**
22      * Declaration of the variables
23     */
24     /* TCP Connect Scanning Thread Parameters Struct */
25     static int TCPConnectThreadNum;
26     static pthread_mutex_t TCPConnectThreadNumMutex; // The mutex for the thread
27     ↳ number
28     static int errorStatus;                         // The global error status for
29     ↳ the single thread
30     static pthread_mutex_t errorStatusMutex;        // The mutex for the error
31     ↳ status
32     static ThreadSafeQueue<LogMessage> logQueue;    // The log queue
33     static pthread_mutex_t logQueueMutex;            // The mutex for the log queue
34
35 public:
36     TCPConnectScanUtil();
37     ~TCPConnectScanUtil();
38     // Mutil Thread Scanning
39     static void *Thread_TCPConnectHost(void *param); // Scanning on the specific
40     ↳ port in this thread
41     static void *Thread_TCPConnectScan(void *param); // Initialize the threads for
42     ↳ scanning, calling upon the Thread_TCPConnectHost function
43 };
44 #endif // TCP_CONNECT_SCAN_HPP

```

---

#### 4.1.5 TCPSynScan.hpp

这个头文件定义了 `TCPSynScanUtil` 工具类，即单独的-s 选项。其中包含了用于 TCP Syn 扫描的相关成员变量和函数声明。其具体变量和函数定义除了名字为 `Thread_TCPSynHost` 之外，其余均和 `TCPConnectScanUtil` 一样。因此这里不再重复展示。

#### 4.1.6 TCPFinScan.hpp

这个头文件定义了 `TCPFinScanUtil` 工具类，即单独的-f 选项。其中包含了用于 TCP Fin 扫描的相关成员变量和函数声明。同样的，其也除了名字不同，其余变量定义均与 `TCPConnectScanUtil` 以及 `TCPSynScanUtil` 一样。因此这里不再重复展示。

#### 4.1.7 UDPScan.hpp

这个头文件定义了 `UDPScanUtil` 工具类，即单独的-u 选项。其中包含了用于 UDP 扫描的相关成员变量和函数声明。**不同的是，这次底层的扫描不再使用多线程函数实现。**

由于目标主机返回的 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。因此，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。这样，判断被扫描端口是开启还是关闭就显得毫无意义了。

为了保证扫描的准确性，必须牺牲程序的运行效率，逐次地扫描目标主机的被测端口。具体声明的代码为：

```
1 // The constants, variables and declarations about the UDP Scanning using UDP
2   → protocol
3
4 #ifndef UDP_SCAN_HPP
5 #define UDP_SCAN_HPP
6
7 #include "def.hpp"
8 #include "utils.hpp"
9 #include <string>           // std::string
10 #include <iostream>         // std::cout, std::endl
11 #include <sys/socket.h>    // socket, AF_INET, SOCK_RAM
12 #include <arpa/inet.h>     // sockaddr_in, inet_addr, htons
13 #include <fcntl.h>          // fcntl, F_SETFL, O_NONBLOCK
14 #include <sys/time.h>        // gettimeofday
15 #include <unistd.h>          // read
16
17 class UDPScanUtil
18 {
19     /**
20      * Declaration of the variables
21     */
22     // Socket Variables
23     static int udpSocket; // socket for UDP
24     static int on;         // the flag for the socket
25
26 public:
27     UDPScanUtil();
28     ~UDPScanUtil();
29     static void *Thread_UDPScan(void *param);
30     static INT UDPScanHost(UDPScanHostThreadParam *param);
31 };
32 #endif // UDP_SCAN_HPP
```

#### 4.1.8 Scanner.hpp

作为程序的主要函数，根据给定选项即 {-h, -c, -s, -f, -u} 依次声明了打印帮助信息，ping 探测程序可达性，TCP Connect 扫描，TCP Syn 扫描，TCP Fin 扫描还有 UDP 扫描的函数。具体代

码如下：

---

```
1 // Declarations of the functions used in Scanner.cpp
2
3 #include "TCPConnectScan.hpp"
4 #include "TCPFinScan.hpp"
5 #include "TCPSynScan.hpp"
6 #include "UDPScan.hpp"
7 #include "ping.hpp"
8 #include <iostream> // std::cout
9
10 inline void help(const char *program);
11 INT ping(std::string ip, int times);
12 inline INT TCPConnectScan(std::string ip, int beginPort, int endPort, INT mode);
13 inline INT TCPSyncan(std::string ip, int beginPort, int endPort, INT mode);
14 inline INT TCPFinScan(std::string ip, int beginPort, int endPort, INT mode);
15 inline INT UDPScan(std::string ip, int beginPort, int endPort, INT mode);
```

---

## 4.2 src

接下来会介绍具体的实现文件。

### 4.2.1 utils.cpp

该文件实现了 utils.hpp 中声明的函数，具体而言：

#### 1. uint16\_t in\_cksum(uint16\_t \*addr, int len)

in\_cksum 的参数包括一个指向网络报头地址的指针 addr 和报头的长度 len。返回值为一个 16 位的校验和值。函数将网络报头的 2 个字节值相加，处理剩余的字节，把 32 位之和折叠到 16 位，并将结果反向计算。

---

```
1 // Compute the checksum of the headers
2 uint16_t in_cksum(uint16_t *addr, int len)
3 {
4     /**
5      * The function to compute the checksum of the headers
6      * @param addr The address of the headers
7      * @param len The length of the headers
8      * @return The checksum of the headers
9      */
10
11     int nleft = len;
12     int sum = 0;
```

```

13     uint16_t *w = addr;
14     uint16_t answer = 0;
15     // Sum up 2-byte values until none or only one byte left.
16     while (nleft > 1)
17     {
18         sum += *w++;
19         nleft -= 2;
20     }
21     // Add leftover byte, if any
22     if (nleft == 1)
23     {
24         *(unsigned char *)&answer = *(unsigned char *)w;
25         sum += answer;
26     }
27     // Fold 32-bit sum into 16 bits
28     sum = (sum >> 16) + (sum & 0xFFFF);
29     sum += (sum >> 16);
30     answer = ~sum;
31
32     return answer;
33 }
```

## 2. bool isValidIPv4(const char \*pszIPAddr)

`isValidIPv4` 参考了博客 [【C++】判断 IP 地址有效性](#) 中的实现，该函数用于检查输入是否为有效的 IPv4 地址。该函数通过检查 IP 地址中的字符和分隔符来验证 IP 地址的有效性，要求 IP 地址由 4 个由“.”分隔的小节组成，每个小节的取值范围为 0 到 255，且不能以 0 开头，IP 地址必须包含 3 个“.”。函数在检查过程中遵循 IPv4 地址的规范，如果输入不符合规则则返回 `false`，否则返回 `true`。

```

1 // Check if the input is a valid IPv4 address
2 bool isValidIPv4(const char *pszIPAddr)
3 {
4     /**
5      * The function to check if the input is a valid IPv4 address
6      * @param pszIPAddr The input IP address
7      * @return True if the input is a valid IPv4 address, otherwise false
8      */
9     if (!pszIPAddr)
10        return false; // If pszIPAddr is null
11
12     char cIP[4];
13     int len = strlen(pszIPAddr);
```

```
14     int n = 0, num = 0;
15
16     const char *p = pszIPAddr;
17
18     while (*p != '\0')
19     {
20         if (*p == ' ' || *p < '0' || *p > '9')
21             return false; // If character is space or not a digit
22
23         cIP[n++] = *p; // Save the first character of the segment to check for
24             // leading zeros
25
26         int sum = 0; // The numeric value of the current segment, must be
27             // between 0 and 255
28         while (*p != '.' && *p != '\0')
29         {
30             if (*p == ' ' || *p < '0' || *p > '9')
31                 return false; // If character is space or not a digit
32             sum = sum * 10 + *p - '0'; // Convert the segment string to an
33             // integer
34             p++;
35         }
36         if (*p == '.')
37         {
38             // Check if there are digits before and after the dot
39             if (((p - 1) >= '0' && *(p - 1) <= '9') && (*(p + 1) >= '0' && *(p
40             + 1) <= '9'))
41                 num++; // Count the number of dots, must be exactly 3
42             else
43                 return false;
44         }
45         if ((sum > 255) || (sum > 0 && cIP[0] == '0') || num > 3)
46             return false; // Invalid if segment value is >255, starts with 0,
47             // or more than 3 dots
48
49         if (*p != '\0')
50             p++;
51         n = 0; // Reset segment character index
52     }
53     if (num != 3)
54         return false; // Must have exactly 3 dots
55     return true;
```

```
51 }
52
```

---

### 3. bool isValidPort(int beginPort, int endPort)

`isValidPort` 实现主要是为了检查输入是否为有效的端口范围。分别以起始端口和终止端口作为函数参数，检查的内容包括起始端口和终止端口是否都在 `MAX_PORT` 和 `MIN_PORT` 范围内，并且起始端口是否大于等于终止端口。函数在检查过程中遵循上述规则，如果输入不符合规则则返回 `false`，否则返回 `true`。

```
1 // Check if the begin and end ports are valid for out of range
2 bool isValidPort(int beginPort, int endPort)
3 {
4     /**
5      * The function to check if the begin and end ports are valid for out of
6      * range
7      * @param beginPort The begin port
8      * @param endPort The end port
9      * @return True if the begin and end ports are valid for out of range,
10      * otherwise false
11 */
12 if (beginPort <= MIN_PORT || beginPort >= MAX_PORT || endPort <= MIN_PORT
13     || endPort >= MAX_PORT || beginPort >= endPort)
14     return false;
15 return true;
16 }
```

---

### 4. void logProcessingThread(ThreadSafeQueue<LogMessage> &logQueue, int beginPort, int endPort)

`logProcessingThread` 函数用于实现安全日志线程的处理。主要做的工作有如下：

- 输出完整日志：通过使用 `logQueue.pop()` 将每一条完整的日志信息提取出来，保证不会输出的日志信息相互混乱。
- 有序输出日志：通过建立 `std::map<int, std::string>` `logMap` 将端口号和字符串建立起一个映射，以端口号为键值按序输出日志信息，保证顺序不会混乱。
- 输出全部日志后再返回：通过 `expectedPort > endPort` 和 `logQueue.empty()` 确保日志队列空了再退出。

```
1 // The log processing thread for the log message
2 void logProcessingThread(ThreadSafeQueue<LogMessage> &logQueue, int beginPort,
3     int endPort)
4 {
```

---

```
4  /**
5   * The log processing thread for the log message
6   * This thread makes sure that the log message is orderly and intactly
7   * → printed according to the port number
8   * @param logQueue The log queue
9   * @param beginPort The begin port
10  * @param endPort The end port
11  */
12 // Define the variables
13 std::map<int, std::string> logMap;
14 int expectedPort = beginPort;
15
16 // The main loop
17 while (true)
18 {
19     // Get one log message from the queue and store it in the map
20     LogMessage log = logQueue.pop();
21     logMap[log.port] = log.message;
22     // Check the map to see if there is any expected log message
23     while (logMap.count(expectedPort))
24     {
25         // Continuously print the log message until the expected port is not
26         // → in the map yet
27         std::cout << logMap[expectedPort] << std::endl;
28         logMap.erase(expectedPort);
29         expectedPort++;
30     }
31
32     if ((expectedPort > endPort) && logQueue.empty())
33         // If the expected port is greater than the end port and the queue
34         // → is empty, then break the loop
35         break;
36 }
37 }
```

#### 4.2.2 ping.cpp

该文件实现了 pings.hpp 中声明的函数，具体而言：

1. **初始化函数:** ICMPUtil(std::string hostIP, int localPort, std::string localHostIP)  
主要进行资源的分配和一些初始化。

---

```

1 ICMPUtil::ICMPUtil(std::string hostIP, int localPort, std::string localHostIP)
2   : hostIP(hostIP), localPort(localPort), localHostIP(localHostIP)
3 {
4   /**
5    * The constructor of the ICMPUtil class
6    * @param hostIP The IP address of the host
7    * @param localPort The port of the local host
8    * @param localHostIP The IP address of the local host
9    */
10  recvBuffer = (char *)malloc(MAX_BUFFERS_SIZE);
11  memset(recvBuffer, 0, MAX_BUFFERS_SIZE);
12  pingtimes = 0;
13  flag = true;
14 }

```

---

## 2. 探测主程序: ping(int times)

该函数接收 `times` 作为探测次数, 0 就是直到超时; 返回 `SUCCESS`, `FAILURE`, `ERROR`, `TIMEOUT`。

- **创建套接字**

使用 `socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)` 函数创建一个原始套接字, 用于发送和接收 ICMP 数据包。

- **设置套接字选项**

通过 `setsockopt()` 函数设置套接字选项, 主要通过 `on=1` 参数设置其自动设置 IP 头部。前两部分代码为:

---

```

1 INT ICMPUtil::ping(int times)
2 {
3   /**
4    * The function to ping the host using ICMP protocol
5    * @param times The times to ping the host
6    *          0 for ping until timeout
7    *          x(x>0) for ping x times
8    * @return The ping results from the host:
9    *          1 for success
10   *          0 for failure
11   *          -1 for error
12   */
13
14 // Create the raw socket
15 pingSocket = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
16

```

---

```

17     /* Set the socket relevant Settings, for the need to include and
18     → operate the IP header */
19     on = 1;
20     ret = setsockopt(pingSocket, 0, IP_HDRINCL, &on, sizeof(on));
21             /* blablabla */
22 }
```

---

- 构建 IP 和 ICMP 请求数据包

为了提高程序的跨平台实用性，我为 ping 程序使用的是非 Linux 平台下也能够有效的 ip 和 icmp 结构体，而不是 iphdr 和 icmphdr，因为后两者只能在 Linux 平台下使用。

构建 IP 和 ICMP 请求数据包，填充 IP 以及 ICMP 头部信息，包括类型、代码、校验和、标识符和序列号等。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。

具体而言，本程序为了实现对 Linux 下 ping 的复刻，ICMP 请求数据包包含的部分有：

- IP 头部 (20 比特)
- ICMP 头部 (8 比特)
- Timeval 结构体记录时间 (16 比特)
- Padding 填充 (40 比特)

总的而言，整体的数据包大小应该为 56 比特（加上头部总共 84 比特）。另外这部分进行的操作还包含地址的赋值，开始计时，调用 in\_cksum 计算校验和等。这部分的代码如下：

---

```

1 INT ICMPUtil::ping(int times)
2 {
3
4     /**
5      * Create the ICMP Echo Package
6      * The ICMP Echo Package should include four parts to meet the standard
7      → ping package on Linux Platform:
8      * 1. IP Header(20 bytes)
9      * 2. ICMP Header(8 bytes)
10     * 3. Timeval struct(16 bytes)
11     * 4. Padding(40 bytes)
12     * So size of the package is 56 bytes(84 bytes in total with the
13     → Header)
14
15     sendBufferSize = sizeof(struct ip) + sizeof(struct icmp) +
16     → sizeof(timeval) + 40;
17     sendBuffer = (char *)malloc(sendBufferSize);
18     memset(sendBuffer, 0, sendBufferSize);
19     memset(recvBuffer, 0, MAX_BUFFERS_SIZE);
20     // Print the PING message
21 }
```

```
18     std::cout << "PING " << hostIP << " (" << hostIP << ") 56(84) bytes of
19     ↳  data." << std::endl;
20
21     // Set the IP Header
22     sendIPHeader = (struct ip *)sendBuffer;
23     sendIPHeader->ip_hl = 5;                                // Header
24     ↳  Length
25     sendIPHeader->ip_v = 4;                                // 
26     ↳  Version, 4 for IPv4
27     sendIPHeader->ip_tos = 0;                               // Type
28     ↳  of Service, 0 for normal
29     sendIPHeader->ip_len = htons(sendBufferSize);          // Total
30     ↳  Length
31     sendIPHeader->ip_id = rand();                           // 
32     ↳  Identification
33     sendIPHeader->ip_ttl = 64;                            // Time
34     ↳  to Live
35     sendIPHeader->ip_off = 0x40;                           // 
36     ↳  Fragment Offset, 0x40 stands for no fragmentation
37     sendIPHeader->ip_p = IPPROTO_ICMP;                     // 
38     ↳  Protocol ICMP
39     sendIPHeader->ip_sum = 0;                             // 
40     ↳  Checksum initially as 0
41
42     sendIPHeader->ip_src.s_addr = inet_addr(localHostIP.c_str()); // Source
43     ↳  IP
44     sendIPHeader->ip_dst.s_addr = inet_addr(hostIP.c_str());    // 
45     ↳  Destination IP
46
47     // Set the ICMP Header
48     sendICMPHeader = (struct icmp *)(sendIPHeader + 1); // ICMP Header
49     ↳  follows the IP Header
50     sendICMPHeader->icmp_type = ICMP_ECHO;                // ICMP Echo
51     ↳  Request
52     sendICMPHeader->icmp_code = 0;                         // Code 0
53     sendICMPHeader->icmp_cksum = 0;                        // Checksum
54     ↳  initially as 0
55     sendICMPHeader->icmp_id = htons(localPort);           // local port as
56     ↳  the ID
57     sendICMPHeader->icmp_seq = htons(1);                  // Sequence Number
58     ↳  usually starts from 1 like the standard ping command
59
60     // Fill the 40 bytes of empty data with 0
```

```

43     memset(sendBuffer + sizeof(struct ip) + sizeof(struct icmp), 0, 40);
44
45     // Compute the checksum of the ICMP Header
46     tp = (struct timeval *) (sendBuffer + sizeof(struct ip) + sizeof(struct
47     ↳ icmp)); // Get the timeval struct
48     gettimeofday(tp, NULL);
49     ↳ // Get the current time
50     sendICMPHeader->icmp_cksum = in_cksum((unsigned short *) sendICMPHeader,
51     ↳ sizeof(struct icmp) + sizeof(struct timeval) + 40);
52
53     // Set the sending address of the socket
54     pingHostAddr.sin_family = AF_INET;                                // IPv4
55     pingHostAddr.sin_addr.s_addr = inet_addr(hostIP.c_str()); // Convert
56     ↳ the IP address to the network byte order
57     addrLen = sizeof(pingHostAddr);
58
59     /* blablabla */
60 }
```

---

- **发送 ICMP 请求数据包**

使用 `sendto()` 函数将 ICMP 请求数据包发送到目标主机。

- **接收 ICMP 响应数据包**

通过 `fcntl` 设置 `socket` 为非阻塞模式。在一个 `while` 循环中忙等待接收信息。若不是第一次接收到信息，每次到循环开头更改 `icmp` 头的序列号 `icmp_seq` 重新计算校验和并发送。

使用 `recvfrom()` 函数接收目标主机返回的数据包。首先判断：

- 源地址 IP 和目标主机 IP 是否一致；
- 目的地址和本机 IP 是否一致；
- ICMP 数据包的 `icmp_type` 字段是否为回显 (`type = 0`, 即为 `ICMP_ECHOREPLY`)。

若符合上述要求，则接收到的就是 ICMP 响应数据包。记录此时经过的时间，并输出收到的比特数，序列号，总长度和经过的时间。输出格式和标准的 `ping` 程序完全一致。

- **处理超时和重传**

因为设置了 `socket` 为非阻塞模式，所以 `recvfrom()` 若未接受到数据包会直接返回。通过记录结束的时间，结合 `tv_sec` 与 `tv_usec` 的级别下判断是否超时。如果在设定的超时时间内没有收到响应数据包，则报告目标主机不可达的超时消息。

以上三部分的代码如下：

---

```

1 INT ICMPUtil::ping(int times)
2 {
3     /* blablabla */
4     // Send the ICMP Request
```

```
5     if (sendto(pingSocket, sendBuffer, sendBufferSize, 0, (struct sockaddr
6         * )&pingHostAddr, addrLen) == -1)
7     {
8         std::cout << "[ERROR] Failed to send the ICMP Request to " <<
9             hostIP << std::endl;
10        // Clear the resources
11        memset(sendBuffer, 0, sendBufferSize);
12        free(sendBuffer);
13        memset(recvBuffer, 0, MAX_BUFFERS_SIZE);
14        free(recvBuffer);
15        close(pingSocket);
16        return ERROR;
17    }
18
19    /* Waiting for ICMP Reply */
20    // Set the socket to non-block mode
21    if (fcntl(pingSocket, F_SETFL, O_NONBLOCK) < 0)
22    {
23        std::cout << "[ERROR] Failed to set the socket to non-blocking mode
24            for ping on ip address " << hostIP << std::endl;
25        // Clear the resources
26        memset(sendBuffer, 0, sendBufferSize);
27        free(sendBuffer);
28        memset(recvBuffer, 0, MAX_BUFFERS_SIZE);
29        free(recvBuffer);
30        close(pingSocket);
31        return ERROR;
32    }
33
34    // Busy waiting for the ICMP Reply
35    gettimeofday(&waitingStartTP, NULL);
36    while (pingtimes < times || times == 0)
37    {
38        if (flag == false)
39        {
40            sleep(1); // Sleep for 2 seconds
41
42            // Update the ICMP header sequence number and timestamp
43            sendICMPHeader->icmp_seq = htons(pingtimes + 1);
44            tp = (struct timeval *) (sendBuffer + sizeof(struct ip) +
45                sizeof(struct icmp));
46            gettimeofday(tp, NULL);
47
48            if (sendto(pingSocket, sendBuffer, sendBufferSize, 0, (struct sockaddr
49                * )&pingHostAddr, addrLen) == -1)
50            {
51                std::cout << "[ERROR] Failed to send the ICMP Request to " <<
52                    hostIP << std::endl;
53                // Clear the resources
54                memset(sendBuffer, 0, sendBufferSize);
55                free(sendBuffer);
56                memset(recvBuffer, 0, MAX_BUFFERS_SIZE);
57                free(recvBuffer);
58                close(pingSocket);
59                return ERROR;
60            }
61        }
62    }
63
64    // Busy waiting for the ICMP Reply
65    gettimeofday(&waitingEndTP, NULL);
66    pingtimes++;
67
68    // Print the ping results
69    std::cout << "Ping statistics for " << hostIP << std::endl;
70    std::cout << "Packets: Sent = " << pingtimes << ", Received = " <<
71        pingtimes << ", Lost = 0 (%)" << std::endl;
72    std::cout << "Round-trip time: " << (waitingEndTP.tv_sec - waitingStartTP.tv_sec) <<
73        " seconds" << std::endl;
74    std::cout << "Throughput: " << (pingtimes * 64 / (waitingEndTP.tv_sec - waitingStartTP.tv_sec)) <<
75        " bytes/second" << std::endl;
76
77    // Clean up
78    memset(sendBuffer, 0, sendBufferSize);
79    free(sendBuffer);
80    memset(recvBuffer, 0, MAX_BUFFERS_SIZE);
81    free(recvBuffer);
82    close(pingSocket);
83
84    return 0;
85}
```

```
43
44     // Recompute the checksum of the ICMP Header
45     sendICMPHeader->icmp_cksum = 0;
46     sendICMPHeader->icmp_cksum = in_cksum((unsigned short
47         *)sendICMPHeader, sizeof(struct icmp) + sizeof(struct
48         timeval) + 40);
49
50     // Send the ICMP Request
51     if (sendto(pingSocket, sendBuffer, sendBufferSize, 0, (struct
52         sockaddr *)&pingHostAddr, addrLen) == -1)
53     {
54         std::cout << "[ERROR] Failed to send the ICMP Request to "
55             << hostIP << std::endl;
56
57         // Clear the resources
58         memset(sendBuffer, 0, sendBufferSize);
59         free(sendBuffer);
60
61         memset(recvBuffer, 0, MAX_BUFFERS_SIZE);
62         free(recvBuffer);
63         close(pingSocket);
64         return ERROR;
65     }
66
67     while (true)
68     {
69         ret = recvfrom(pingSocket, recvBuffer, MAX_BUFFERS_SIZE, 0,
70             (struct sockaddr *)&fromHostAddr, (socklen_t *)&addrLen);
71         if (ret > 0)
72         {
73             /**
74              * If the recvfrom returns a positive value, one reply is
75              * received with its actual size
76              * Then check the corresponding requirements:
77              *      (1) If the source IP is the same as the host IP
78              *      (2) If the destination IP is the same as the local
79              *          host IP
80         }
```

```
76          *      (3) If the ICMP header with type code:  
77          ↳  ICMP_ECHOREPLY  
78          */  
79          // Get its IP Header  
80          recvIPHeader = (struct ip *)recvBuffer;  
81          recvICMPHeader = (struct icmp *)(recvBuffer +  
82          ↳  (recvIPHeader->ip_hl << 2));  
83          // Get the source and destination IP  
84          sourceHostIP = inet_ntoa(recvIPHeader->ip_src);  
85          destHostIP = inet_ntoa(recvIPHeader->ip_dst);  
86  
87          // Check the requirements  
88          if (sourceHostIP == hostIP &&  
89          destHostIP == localHostIP &&  
90          recvICMPHeader->icmp_type == ICMP_ECHOREPLY)  
91          {  
92              gettimeofday(&waitingEndTP, NULL);  
93              double elapsedTime = (waitingEndTP.tv_sec -  
94              ↳  waitingStartTP.tv_sec) * 1000.0;  
95              elapsedTime += (waitingEndTP.tv_usec -  
96              ↳  waitingStartTP.tv_usec) / 1000.0;  
97  
98              std::cout << ret << " bytes from " << hostIP << ":"  
99              ↳  icmp_seq=" << ntohs(recvICMPHeader->icmp_seq) << "  
100             ↳  ttl=" << (int)recvIPHeader->ip_ttl << " time=" <<  
101             ↳  elapsedTime << " ms" << std::endl;  
102             pingtimes++;  
103             flag = false;  
104             break;  
105         }  
106     }  
107     // If the control flow reaches here, due to the non-block mode  
108     ↳  of the socket  
109     // Check the time to avoid the infinite loop  
110  
111     gettimeofday(&waitingEndTP, NULL);  
112     if ((1000000 *  
113         (waitingEndTP.tv_sec - waitingStartTP.tv_sec) +  
114         (waitingEndTP.tv_usec - waitingStartTP.tv_usec)) /  
115         1000000.0 >  
116         MAX_PING_TIMEOUT)  
117     return TIMEOUT;
```

```

110         }
111     }
112
113     // Clear the resources
114     memset(sendBuffer, 0, sendBufferSize);
115     free(sendBuffer);
116     memset(recvBuffer, 0, MAX_BUFFERS_SIZE);
117     free(recvBuffer);
118     close(pingSocket);
119     return SUCCESS;
120 }
```

---

上述过程中对任何错误情况均进行了处理，具体而言就是输出错误提示日志信息，清理资源等避免错误。比如代码：

```

1          /* blablabla */
2
3      // Check the return value
4      if (ret == -1)
5      {
6          std::cout << "[ERROR] Failed to set the socket option for ping on ip address"
7          << " " << hostIP << std::endl;
8
9      // Clear the resources
10     memset(sendBuffer, 0, sendBufferSize);
11     free(sendBuffer);
12     memset(recvBuffer, 0, MAX_BUFFERS_SIZE);
13     free(recvBuffer);
14     close(pingSocket);
15
16     return ERROR;
17
18     /* blablabla */
19 }
```

---

#### 4.2.3 TCPConnectScan.cpp

该文件实现了 `TCPConnectScan.hpp` 中声明的函数和一些变量，具体而言：

##### 1. 静态变量定义

对 `TCPConnectThreadNum`, `logQueue` 等头文件中声明的静态变量进行定义。方便后续使用。

##### 2. 范围扫描线程函数 `void *TCPConnectScanUtil::Thread_TCPConnectScan(void *param)`

该函数负责调用底层的 `Thread_TCPConnectHost` 对某个特定端口进行扫描。具体而言，其主要做了以下内容：

- 提取参数与初始化信息

该函数首先通过上层（调用其的 `Scanner.cpp`）传递过来的 `TCPConnectThreadParam` 提取要扫描的主机地址和起始、终止端口。并初始化一些变量，并启动线程安全的日志线程，日志线程主函数为 `logPreocessingThread`。这部分代码为：

```
1 // Initialize the threads for scanning, calling upon the
2   ↳ Thread_TCPConnectHost function
3 void *TCPConnectScanUtil::Thread_TCPConnectScan(void *param)
4 {
5   /**
6    * The advanced function to TCP scan for the range of ports, calling
7    ↳ upon the Thread_TCPConnectHost function
8    * Using multithreading to improve the performance
9    * @param param: The port range
10   * @return: SUCCESS or ERROR
11 */
12
13 // Declare the variables
14 struct TCPConnectThreadParam *p;
15 std::string hostIP;
16 int localPort;
17 int beginPort;
18 int tempPort; // The temporary port for scanning
19 int endPort;
20 pthread_attr_t attr;
21 pthread_t childThreadID;
22 pthread_t loggerThreadID;
23 INT ret;
24 int scanFailureFlag;
25
26 // Get the parameters from the struct
27 p = (struct TCPConnectThreadParam *)param;
28 hostIP = p->hostIP;
29 beginPort = p->beginPort;
30 endPort = p->endPort;
31
32 // Set the number of threads to 0
33 TCPConnectThreadNum = 0;
34 errorStatus = SUCCESS;
35 scanFailureFlag = false;
36
37 std::cout << "[INFO] Begin TCP connect scan..." << std::endl;
```

```

36
37     // Create the log processing thread
38     ret = pthread_create(&loggerThreadID, NULL, [](void *arg) -> void *
39     {
40         auto* args = static_cast<std::tuple<ThreadSafeQueue<LogMessage>*>,
41             & int, & int>*(arg);
42         logProcessingThread(*std::get<0>(*args), std::get<1>(*args),
43             & std::get<2>(*args));
44         delete args;
45         return nullptr; }, new std::tuple<ThreadSafeQueue<LogMessage> *, 
46             & int, & int>(&logQueue, beginPort, endPort));
47             /* blablabla */
48 }

```

---

- **循环遍历端口调用底层多线程函数**

这部分通过一个 `for` 循环，利用局部变量 `tempPort` 指定要扫描的端口。对每一个要扫描的端口通过 `TCPConnectHostThreadParam` 结构体赋予参数值，设置线程为 `PTHREAD_CREATE_DETACHED` 即脱离状态方便资源释放。创建每一个子线程执行函数 `Thread_TCPConnectHost` 扫描 `tempPort`。并打开 `mutex` 锁设置 `TCPConnectThreadNum` 自增。若线程过多了，会停止 3 秒避免资源被过度浪费。

```

1 Thread_TCPConnectHost function
2 void *TCPConnectScanUtil::Thread_TCPConnectScan(void *param)
3 {
4
5     /* blablabla */
6     // Loop through the range of ports
7     for (tempPort = beginPort; tempPort <= endPort; tempPort++)
8     {
9
10        // Set the parameters for the child thread
11        TCPConnectHostThreadParam *pConHostParam = new
12            & TCPConnectHostThreadParam();
13        pConHostParam->hostIP = hostIP;
14        pConHostParam->port = tempPort;
15
16        // Set the child thread separated
17        pthread_attr_init(&attr);
18        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
19        // Create the child thread

```

```
20     ret = pthread_create(&childThreadID, &attr, Thread_TCPConnectHost,
21                           (void *)pConHostParam);
22
23     if (ret != 0)
24     {
25         // Error
26         std::cerr << "[ERROR] Failed to create the thread for the TCP
27             Connect Scanning on ip address " << hostIP << " and port "
28             << tempPort << std::endl;
29         scanFailureFlag = true;
30         break;
31     }
32
33     // Increase the number of threads
34     pthread_mutex_lock(&TCPConnectThreadNumMutex);
35     TCPConnectThreadNum++;
36     pthread_mutex_unlock(&TCPConnectThreadNumMutex);
37     // In order to avoid too many threads, we need to wait for a while
38     while (TCPConnectThreadNum > 100)
39         sleep(3);
40
41     // If the control flow reaches here, the scanning of the range of ports
42     // has been successful assigned to the threads
43     // Wait for the threads to finish
44     while (TCPConnectThreadNum != 0)
45     {
46         // Check the error status
47         pthread_mutex_lock(&errorStatusMutex);
48         if (errorStatus == ERROR)
49         {
50             // Error
51             pthread_mutex_unlock(&errorStatusMutex);
52             scanFailureFlag = true;
53         }
54         pthread_mutex_unlock(&errorStatusMutex);
55         sleep(1);
56     }
57
58     /* blablabla */
59 }
```

- 线程收尾与资源清理

这部分检测 `TCPConnectThreadNum` 何时为 0，若为 0，则代表所有线程都退出了，主线程也可以退出了。在这个循环中锁住 `mutex` 并检测其它线程会赋值的 `errorStatus`。若发生错误，最后退出时候标记为 `pthread_exit((void *)ERROR)`，否则标记 `SUCCESS`。

---

```

1 Thread_TCPConnectHost function
2 void *TCPConnectScanUtil::Thread_TCPConnectScan(void *param)
3 {
4
5             /* blablabla */
6
7     // If the control flow reaches here, the scanning of the range of ports
8     // has been successful assigned to the threads
9     // Wait for the threads to finish
10    while (TCPConnectThreadNum != 0)
11    {
12        // Check the error status
13        pthread_mutex_lock(&errorStatusMutex);
14        if (errorStatus == ERROR)
15        {
16            // Error
17            pthread_mutex_unlock(&errorStatusMutex);
18            scanFailureFlag = true;
19        }
20        pthread_mutex_unlock(&errorStatusMutex);
21        sleep(1);
22    }
23    // Wait for the logger thread to finish
24    pthread_join(loggerThreadID, NULL);
25
26    // The main thread exits
27    if (scanFailureFlag)
28        // Error
29        pthread_exit((void *)ERROR);
30    else
31        // Success
32        pthread_exit((void *)SUCCESS);
33 }
```

---

3. 指定端口扫描线程函数 `void *TCPConnectScanUtil::Thread_TCPConnectHost(void *param)`  
该函数实现比较简单，具体而言：

- (a) 创建套接字并设置地址

最开始也是通过 `TCPConnectHostThreadParam` 获取参数即指定的地址和端口。使用 `socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)` 函数创建一个 TCP 套接字，用于发送和接收 TCP 数据包。设置 `hostAddress`。

(b) 尝试连接目标端口

使用 `connect()` 函数尝试连接目标主机的指定端口。扫描发起主机发送一个 SYN 包给目标主机的指定端口。

(c) 判断端口状态

根据 `connect()` 函数的返回值来判断端口状态。如果返回值为 0，表示连接成功，端口开放；如果返回值为 -1，表示连接失败，端口关闭。锁住 `mutex`，不是将消息直接打印，而是作为日志信息暂时存取起来。

(d) 资源清理并存放日志信息

最后将 `socket` 关闭，日志信息压入日志队列交给日志线程处理。最后将 `TCPConnectThreadNum` 自减后退出。

完整代码为：

```
1 // Scan on the specific port in this thread
2 void *TCPConnectScanUtil::Thread_TCPConnectHost(void *param)
3 {
4     /**
5      * The base function to TCP scan for the specific port
6      * Using multithreading to improve the performance
7      * @param param: The port number
8      * @return: WHATEVER, due to the detached state of the threads
9      */
10    // Declare the variables
11    struct TCPConnectHostThreadParam *p;
12    std::string hostIP;
13    int localPort;
14    int port;
15    int connectSocket;
16    struct sockaddr_in hostAddress;
17    INT ret;
18    std::string logMessage;
19
20    // Get the parameters from the struct
21    p = (struct TCPConnectHostThreadParam *)param;
22    hostIP = p->hostIP;
23    port = p->port;
24
25    // Create socket
26    connectSocket = socket(AF_INET, SOCK_STREAM, 0);
```

```
27     // Check the socket
28     if (connectSocket < 0)
29     {
30         // Error
31         std::cerr << "[ERROR] Failed to create the socket for the TCP Connect"
32             ↳ Scanning on ip address " << hostIP << " and port " << port <<
33             ↳ std::endl;
34         pthread_mutex_lock(&TCPConnectThreadNumMutex);
35         errorStatus = ERROR;
36         pthread_mutex_unlock(&errorStatusMutex);
37
38         // Clear the resources
39         delete p;
40         close(connectSocket);
41         // Decrease the number of threads
42         pthread_mutex_lock(&TCPConnectThreadNumMutex);
43         TCPConnectThreadNum--;
44         pthread_mutex_unlock(&TCPConnectThreadNumMutex);
45
46         pthread_exit((void *)ERROR);
47     }
48
49     // Set the host address
50     memset(&hostAddress, 0, sizeof(hostAddress));
51     hostAddress.sin_family = AF_INET;
52     hostAddress.sin_port = htons(port);
53     hostAddress.sin_addr.s_addr = inet_addr(hostIP.c_str());
54
55     // Connect to the host
56     ret = connect(connectSocket, (struct sockaddr *)&hostAddress,
57             ↳ sizeof(hostAddress));
58
59     // Check the connection
60     if (ret < 0)
61     {
62         // Error
63         if (errno == ECONNREFUSED)
64             // When the connection is refused, the port is closed or filtered,
65             ↳ but it can't to determine which one
66             // std::cout << "[INFO] Host: " << hostIP << " Port: " << port <<
67             ↳ closed or filtered!" << std::endl;
```

```
63         logMessage = "[INFO] Host: " + hostIP + " Port: " +
    ↵     std::to_string(port) + " closed or filtered!";
64     else
65     {
66         pthread_mutex_lock(&TCPConnectThreadNumMutex);
67         errorStatus = ERROR;
68         pthread_mutex_unlock(&errorStatusMutex);
69         // std::cerr << "[ERROR] Failed to connect to host: " << hostIP <<
    ↵     " Port: " << port << ". Error: " << strerror(errno) <<
    ↵     std::endl;
70         logMessage = "[ERROR] Failed to connect to host: " + hostIP + " "
    ↵     " Port: " + std::to_string(port) + ". Error: " + strerror(errno);
71     }
72 }
73 else
74     // std::cout << "[INFO] Host: " << hostIP << " Port: " << port <<
    ↵     " open!" << std::endl;
75 logMessage = "[INFO] Host: " + hostIP + " Port: " +
    ↵     std::to_string(port) + " open!";
76
77 // Clear the resources
78 delete p;
79 close(connectSocket);
80
81 // Push the log message to the queue
82 LogMessage log = {port, logMessage};
83 pthread_mutex_lock(&logQueueMutex);
84 logQueue.push(log);
85 pthread_mutex_unlock(&logQueueMutex);
86
87 // Decrease the number of threads
88 pthread_mutex_lock(&TCPConnectThreadNumMutex);
89 TCPConnectThreadNum--;
90 pthread_mutex_unlock(&TCPConnectThreadNumMutex);
91
92 // Exit the thread
93 pthread_exit(NULL);
94 }
```

#### 4.2.4 TCPSynScan.cpp

该文件实现了 `TCPSynScan.hpp` 中声明的函数和一些变量，由于其大体逻辑和 `TCPConnectScan` 的逻辑类似，因此后面只提其不同的地方。详细介绍请参照提供的代码文件。

##### 1. 静态变量定义

##### 2. 范围扫描线程函数 `void *TCPSynScanUtil::Thread_TCPSynScan(void *param)`

该函数负责调用底层的 `Thread_TCPSynHost` 对某个特定端口进行扫描。具体而言，其主要做了以下内容：

###### (a) 提取参数与初始化信息

###### (b) 循环遍历端口调用底层多线程函数

这部分和 TCP Connect Scan 几乎一致，只是构造的参数为 `TCPSynHostThreadParam`，启动的线程函数为：`Thread_TCPSynHost`

###### (c) 线程收尾与资源清理

这部分也和 TCP Connect Scan 几乎一致。不再重复赘述。

##### 3. 指定端口扫描线程函数 `void *TCPSynScanUtil::Thread_TCPSynHost(void *param)`

该函数和 TCP Connect Scan 的线程扫描函数有着一些区别，具体而言：

###### (a) 创建套接字并设置地址

最开始也是通过 `TCPSynHostThreadParam` 获取参数即指定的地址和端口。但是使用的 `socket` 为 `socket(AF_INET, SOCK_RAW, IPPROTO_TCP)`。即原始套接字，方便后续设置 TCP 头部。

###### (b) 构建 SYN 数据包

构建 SYN 数据包，填充 TCP 头部信息，包括源端口、目标端口、序列号、校验和等。其中比较重要的步骤是 `tcpiph->th_flags = TH_SYN`。

###### (c) 发送 SYN 数据包

使用 `sendto()` 函数将 SYN 数据包发送到目标主机的指定端口。

---

```

1 // The function for scanning on the specific port in this thread
2 void *TCPSynScanUtil::Thread_TCPSynHost(void *param)
3 {
4     /**
5      * The function for scanning on the specific port in this thread
6      * @param param: The parameters for the scanning
7      * @return: WHATEVER
8     */
9     /* blablabla */
10    // Memset the TCP SYN Package
11    ptcph = (struct pseudohdr *)sendBuffer;
12    tcpiph = (struct tcphdr *)(sendBuffer + sizeof(struct pseudohdr));

```

```

13
14    // Memset the psedo TCP Header
15    // To calculate the checksum of the TCP package
16    ptcph->saddr = inet_addr(localHostIP.c_str());
17    ptcph->daddr = inet_addr(hostIP.c_str());
18    ptcph->useless = 0; // To fill the 8 bits of 0
19    ptcph->protocol = IPPROTO_TCP;
20    ptcph->length = htons(sizeof(struct tcphdr));
21
22    // Memset the TCP Header
23    tcpiph->th_sport = htons(localPort); // Source port
24    tcpiph->th_dport = htons(port);      // Destination port
25    tcpiph->th_seq = htons(123456);       // Sequence number
26    tcpiph->th_ack = 0;                  // Acknowledgement number
27    tcpiph->th_x2 = 0;                  // Unused
28    tcpiph->th_off = 5;                 // Data offset
29    tcpiph->th_flags = TH_SYN;          // Flags to set it as SYN
30    tcpiph->th_win = htons(65535);       // Window size
31    tcpiph->th_sum = 0;                 // Checksum initialized to 0
32    tcpiph->th_urp = 0;                // Urgent pointer
33
34    // Compute the cksum using in_cksum
35    // 12(Pseudo TCP Header) + 20(TCP Header)
36    tcpiph->th_sum = in_cksum((unsigned short *)ptcph, 20 + 12);
37
38    // Send the SYN package without the pseudo header
39    len = sendto(synSocket, tcpiph, 20, 0, (struct sockaddr
40        *)&SYNScanHostAddress, sizeof(SYNScanHostAddress));
41        /* blablabla */
41 }

```

---

(d) 接收响应数据包

(e) 判断端口状态

根据接收到的数据包判断目标和本地的端口地址是否一致。如果收到 ACK|SYN 数据包，表示端口开放；如果收到 RST 数据包，表示端口关闭；如果未收到任何数据包，则可能被防火墙过滤。

---

```

1 // The function for scanning on the specific port in this thread
2 void *TCPsynScanUtil::Thread_TCPSynHost(void *param)
3 {
4     /**

```

```
5      * The function for scanning on the specific port in this thread
6      * @param param: The parameters for the scanning
7      * @return: WHATEVER
8      */
9      /* blablabla */
10     while (true)
11     {
12         // Take the received Package as a file descriptor, using ssize_t
13         → read (int fd, void *buf, size_t count)
14         len = read(synSocket, recvBuffer, MAX_BUFFERS_SIZE);
15         // Check the read results
16         if (len > 0)
17         {
18             // If the control flow reaches here, the package is received
19             /* Parse the received Package */
20             // Get the IP Header and TCP Header
21             recvIPHeader = (struct iphdr *)recvBuffer;
22             recvTCPHeader = (struct tcphdr *)(recvBuffer +
23             → (recvIPHeader->ihl << 2));
24
25             // Get the source and destination IP address
26             sourceHostIP = inet_ntoa(*(struct in_addr
27             → *)&recvIPHeader->saddr);
28             destHostIP = inet_ntoa(*(struct in_addr
29             → *)&recvIPHeader->daddr);
30             // Get the source and destination port
31             sourcePort = ntohs(recvTCPHeader->source);
32             destPort = ntohs(recvTCPHeader->dest);
33
34             /**
35             * Check if the requirements are met:
36             * (1) Check if the source/dest IP and Port matches
37             * (2) Check if the package type if SYN/ACK
38             */
39             // Check if the source/dest IP and Port matches
40             if (sourceHostIP == hostIP && destHostIP == localHostIP &&
41             → sourcePort == port && destPort == localPort)
42             {
43                 // Debug
44                 // std::cout << "[INFO] th_flags: 0x" << std::hex <<
45                 → static_cast<int>(recvTCPHeader->th_flags) << std::dec
46                 → << std::endl;
```

```

40         // Check if the package type is SYN/ACK
41         if ((recvTCPHeader->th_flags & (TH_SYN | TH_ACK)) ==
42             (TH_SYN | TH_ACK))
43         {
44             // The port is open
45             logMessage = "[INFO] Host: " + hostIP + " Port: " +
46                 std::to_string(port) + " open!";
47             break;
48         }
49         else if ((recvTCPHeader->th_flags & TH_RST) == TH_RST)
50         {
51             // If the received package type includes TH_RST, the
52             // port is closed
53             logMessage = "[INFO] Host: " + hostIP + " Port: " +
54                 std::to_string(port) + " closed!";
55             break;
56         }
57     }

```

---

## (f) 处理超时和重传

## (g) 资源清理并存放日志信息

**4.2.5 TCPFinScan.cpp**

该文件实现了 `TCPFinScan.hpp` 中声明的函数和一些变量，由于其大体逻辑和 `TCPsynScan` 的逻辑几乎完全一致，因此这里只提一些不同的地方。详细介绍请参照提供的代码文件。

1. `tcph->th_flags = TH_FIN;`
2. `Thread_TCPFinHost` 中若收到 RST 标志位则端口开放，若超时没有响应，则端口关闭。

**4.2.6 UDPScan.cpp**

该文件实现了 `UDPScan.hpp` 中声明的函数和一些变量，由于其大体逻辑和上述几个扫描的逻辑比较类似，这里同样只提一些不同的地方。详细介绍请参照提供的代码文件。

1. `UDP Packet = IP Header + UDP Header。` 同时这段通过 `pseudoHeader` 计算 UDP 头部校验和
2. 手动设置 IP 头部为 `ipHeader->protocol = IPPROTO_UDP`。

```
1 // The base function to UDP scan for the specific port
2 INT UDPScanUtil::UDPScanHost(UDPScanHostThreadParam *param)
3 {
4     /* blablabla */
5     // Memset the UDP packet
6     packetSize = sizeof(struct iphdr) + sizeof(struct udphdr);
7     packet = (char *)malloc(packetSize);
8     memset(packet, 0, packetSize);
9
10    /**
11     * Create the UDP Packet
12     * The IUDP Package should include two parts to meet the standard UDP
13     ← package on Linux Platform:
14     * 1. IP Header
15     * 2. UDP Header
16     * Because the pseudo header is only used to calculate the checksum of the
17     ← udp
18     * So there is no need to include the pseudo header in the packet
19     */
20
21    // Create the UDP Packet
22    ipHeader = (struct iphdr *)packet;
23    udpHeader = (struct udphdr *)(packet + sizeof(struct iphdr));
24    pseudoHeader = (struct pseudohdr *)(packet + sizeof(struct iphdr) -
25        → sizeof(struct pseudohdr));
26
27    // Memset the UDP Header
28    udpHeader->source = htons(localPort);           // The source port is the
29    ← local port
30    udpHeader->dest = htons(port);                  // The destination port is
31    ← the scanning port
32    udpHeader->len = htons(sizeof(struct udphdr)); // The length of the UDP
33    ← header
34    udpHeader->check = 0;                           // The checksum of the UDP
35    ← header initially set to 0
36
37    // Memset the Pseudo UDP Header, to calculate the checksum of the UDP
38    ← packet
39    pseudoHeader->saddr = inet_addr(localHostIP.c_str()); // The source IP
40    ← address
41    pseudoHeader->daddr = inet_addr(hostIP.c_str());      // The destination IP
42    ← address
```

```

32     pseudoHeader->useless = 0;                                // To fill the 8 bits
    ↵   of 0
33     pseudoHeader->protocol = IPPROTO_UDP;                  // The protocol type
    ↵   is UDP
34     pseudoHeader->length = udpHeader->len;                 // The length of the
    ↵   UDP package
35
36     // Calculate the checksum of the UDP packet
37     udpHeader->check = in_cksum((unsigned short *)pseudoHeader, sizeof(struct
    ↵   udphdr) + sizeof(struct pseudohdr));
38
39     // Memset the IP Header
40     ipHeader->ihl = 5;                                       // Header Length
41     ipHeader->version = 4;                                     // Version, 4 for IPv4
42     ipHeader->tos = 0x10;                                     // Type of Service, 0x10
    ↵   for IPTOS_LOWDELAY
43     ipHeader->tot_len = htons(packetSize);                  // Total Length is the
    ↵   size of the packet
44     ipHeader->id = htons(54321);                            // Identification
45     ipHeader->frag_off = 0;                                   // Fragment Offset, 0
    ↵   stands for no fragmentation
46     ipHeader->ttl = 69;                                      // Time to Live
47     ipHeader->protocol = IPPROTO_UDP;                      // Protocol UDP
48     ipHeader->check = 0;                                     // Checksum initially as
    ↵   0
49     ipHeader->saddr = inet_addr(localHostIP.c_str()); // Source IP
50     ipHeader->daddr = inet_addr(hostIP.c_str());          // Destination IP
51
52     // Calculate the IP header checksum
53     ipHeader->check = in_cksum((unsigned short *)ipHeader, sizeof(struct
    ↵   iphdr));
54
55     // Send the UDP packet
56     len = sendto(udpSocket, packet, packetSize, 0, (struct sockaddr
    ↵   *)&UDPScanHostAddr, addrLen);
57                                         /* blablabla */
58 }

```

3. 忙等待时候，若接收到对应的数据包，应进行如下判断：

- 源地址是否是目的地址
- 目的地址是否是本机地址

- ICMP 头部 code 字段是否为 ICMP\_DEST\_UNREACH 3
- ICMP 头部 type 字段是否为 ICMP\_PORT\_UNREACH 3

```
1 // The base function to UDP scan for the specific port
2 INT UDPScanUtil::UDPScanHost(UDPScanHostThreadParam *param)
3 {
4     /* blablabla */
5     // Busy waiting for the ICMP Reply
6     while (true)
7     {
8         // Receive the ICMP Packet
9         len = read(udpSocket, (struct ipicmphdr *)&recvHeader,
10             ↳ sizeof(recvHeader));
11         if (len > 0)
12         {
13             /**
14             * If the recvfrom returns a positive value, one reply is received
15             ↳ with its actual size
16             *
17             * Then check the corresponding requirements:
18             *
19             * (1) If the source IP is the same as the host IP
20             * (2) If the destination IP is the same as the local host IP
21             * (3) If the ICMP header with code: ICMP_DEST_UNREACH 3
22             * (4) If the ICMP header with type: ICMP_PORT_UNREACH 3
23             */
24             if (
25                 recvHeader->ipHeader.saddr == inet_addr(hostIP.c_str()) &&
26                 recvHeader->ipHeader.daddr == inet_addr(localHostIP.c_str()) &&
27                 recvHeader->icmpHeader.code == ICMP_DEST_UNREACH &&
28                 recvHeader->icmpHeader.type == ICMP_PORT_UNREACH)
29             {
30                 // Output the log message
31                 std::cout << "[INFO] Host: " + hostIP + " Port: " +
32                     ↳ std::to_string(port) + " closed!" << std::endl;
33                 break;
34             }
35         }
36
37         // If the control flow reaches here, due to the non - block mode of the
38         ↳ socket
39         // Check the time to avoid the infinite loop
40         gettimeofday(&waitingEndTP, NULL);
41         if ((1000000 *
42             ↳ waitingEndTP.tv_sec - waitingStartTP.tv_sec) >= 1000000)
```

```

36             (waitingEndTP.tv_sec - waitingStartTP.tv_sec) +
37             (waitingEndTP.tv_usec - waitingStartTP.tv_usec)) /
38             1000000.0 >
39             MAX_UDP_TIMEOUT)
40     {
41         // Timeout without receiving the ICMP Packet
42         // But due to the previous success ping results, the port is OPEN
43         // without any reply
44         std::cout << "[INFO] Host: " + hostIP + " Port: " +
45             std::to_string(port) + " open or filtered!" << std::endl;
46         break;
47     }
48 }
```

---

#### 4.2.7 Scanner.cpp

`Scanner.cpp` 是整个实验的主程序, 主要在 `main` 函数中负责通过解析两个参数 `int argc, char *argv[]` 进一步进行扫描。其中对 `Scanner.hpp` 实现的主要函数如下:

1. INT ping(std::string ip, int times)

直接初始化 `icmpUtil` 工具类, 传入默认的本地网络参数后启动 `ping` 扫描。根据返回结果不同输出不同日志信息。

---

```

1 INT ping(std::string ip, int times)
2 {
3     /**
4      * @brief Ping the target IP address to check if it is alive
5      * @param ip the target IP address
6      * @param times the times to ping the target IP address
7      * @return The ping results from the host:
8      *         SUCCESS(1) for success
9      *         FAILURE(0) for failure
10     *        ERROR(-1) for error
11     *        TIMEOUT(-2) for timeout
12     */
13
14     // Ping the target IP address
15     ICMPUtil icmpUtil(ip, DEFAULT_LOCAL_PORT, DEFAULT_LOCAL_INET_IP);
16     std::cout << "----- Ping Target IP Address" << std::endl;
17     INT ret = icmpUtil.ping(times);
18     if (ret == SUCCESS)
```

```

19     {
20         std::cout << "[INFO] Ping the target IP address " << ip <<
21             " successfully " << std::endl;
22         std::cout
23             <<
24             "-----"
25             << std::endl;
26         return SUCCESS;
27     }
28     /* 其它错误处理 */
29
30 }

```

---

## 2. inline void help(const char \*program)

根据传入的 `const char *program` 解析程序名字，并打印帮助信息。这里就不展示代码了，详见代码文件或者下面的帮助信息实验结果演示。

## 3. inline INT TCPConnectScan(std::string ip, int beginPort, int endPort, INT mode)

以 `TCPConnectScan` 为例，`TCPSynScan`，`TCPFinScan` 和 `UDPScan` 都代码几乎一致，不再展示。

主要根据传入的参数 `INT mode` 判断是否需要手动输入端口。而后先调用 `ping(ip, DEFAULT_PING_TIMES)` 探测主机可达性 3 次，要是无法打通直接返回。而后初始化线程，调用 `TCPConnectScanUtil::Thread_TCPConnectScan` 该对应扫描工具类的范围扫描函数。最后根据 `pthread_join` 的返回信息判断是否出错。

```

1 inline INT TCPConnectScan(std::string ip, int beginPort, int endPort, INT mode)
2 {
3     /**
4      * @brief TCP Connect scan at the target address from the begin port to the
5          end port
6      * @param ip the target IP address
7      * @param beginPort the begin port
8      * @param endPort the end port
9      * @param mode the mode to input the begin port and end port
10         *           MANUAL(0) for manually input
11         *           AUTO(1) for automatically input
12     */
13     // Declare the variables
14     INT ret;
15     pthread_attr_t attr;
16     pthread_t childThreadID;
17     struct TCPConnectThreadParam param;
18     int status;
19
20 }

```

```
18
19     std::cout << "----- TCP Connect Scan"
20     ↵ -----" << std::endl;
21
22     if (mode == MANUAL)
23     {
24         // If the mode is manual, input the begin port and end port
25         std::cout << "[INFO] Please input the range of port(0-65535)..." <<
26         ↵ std::endl;
27         std::cout << "Begin port: ";
28         std::cin >> beginPort;
29         std::cout << "End port: ";
30         std::cin >> endPort;
31
32         // Check the port range to see if it is valid
33         if (!isValidPort(beginPort, endPort))
34         {
35             std::cerr << "[ERROR] The begin port and end port are INVALID" <<
36             ↵ std::endl;
37             return FAILURE;
38         }
39
40     /* mode == AUTO */
41     // If the mode if AUTO, it means that the port range has already been given
42     ↵ by the CML arguments
43     // Ping the target IP address first to make sure it is alive
44     ret = ping(ip, DEFAULT_PING_TIMES);
45     // If the ping is failed, return the error directly and stop the scanning
46     if (ret == FAILURE || ret == ERROR || ret == TIMEOUT)
47         return ret;
48
49     // Begin the TCP Connect Scan
50     std::cout << "[INFO] TCP Connect Scan Host " << ip << " port " << beginPort
51     ↵ << "~" << endPort << "..." << std::endl;
52
53     // Assign the struct for the parameters
54     param.hostIP = ip;
55     param.beginPort = beginPort;
56     param.endPort = endPort;
57
58     // Initialize the threads for scanning, calling upon the
59     ↵ Thread_TCPConnectScan function
60     pthread_attr_init(&attr);
```

```
54     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
55
56     // Create the child thread
57     ret = pthread_create(&childThreadID, &attr,
58                         TCPConnectScanUtil::Thread_TCPConnectScan, (void *)&param);
59
60     // Check if the thread is created successfully
61     if (ret != 0)
62     {
63         // Error
64         std::cerr << "[ERROR] Failed to create the thread for the TCP Connect
65             Scanning on ip address " << ip << " and port " << beginPort << "~"
66             << endPort << std::endl;
67         return FAILURE;
68     }
69
70     // Wait for the child thread to finish
71     pthread_join(childThreadID, (void **)&status);
72
73     // Check the status of the thread
74     if (status == SUCCESS)
75     {
76         std::cout << "[INFO] TCP Connect Scan Host " << ip << " port " <<
77             beginPort << "~" << endPort << " successfully" << std::endl;
78         std::cout <<
79             "-----"
80             << std::endl;
81         return SUCCESS;
82     }
83     else
84     {
85         std::cerr << "[ERROR] TCP Connect Scan Host " << ip << " port " <<
86             beginPort << "~" << endPort << " FAILED" << std::endl;
87         std::cout <<
88             "-----"
89             << std::endl;
90         return FAILURE;
91     }
92 }
93 return SUCCESS;
94 }
```

### 4.3 Makefile

由于本次实验实现文件较多，即使通过 VSCode 的 .vscode 文件搭建也比较麻烦，因此采用了 makefile 编译。将目标文件生成在 obj 之下，include 下是头文件，src 下是实现文件。具体 makefile 如下：

```
1 # 编译器
2 CXX = g++-11
3 # 编译器标志
4 CXXFLAGS = -g -Wall -std=c++11 -I include
5 # 链接标志
6 LDFLAGS = -lgtest -lpthread -lrt
7 # 目标可执行文件
8 TARGET = bin/Scanner
9 # 源文件目录
10 SRCDIR = src
11 # 目标文件目录
12 OBJDIR = obj
13 # 源文件
14 SRCS = $(SRCDIR)/Scanner.cpp $(SRCDIR)/ping.cpp $(SRCDIR)/TCPConnectScan.cpp \
15      $(SRCDIR)/TCPFinScan.cpp $(SRCDIR)/TCPSynScan.cpp $(SRCDIR)/UDPScan.cpp \
16      $(SRCDIR)/utils.cpp
17 # 目标文件
18 OBJS = $(patsubst $(SRCDIR)/%.cpp,$(OBJDIR)/%.o,$(SRCS))
19
20 # 默认目标
21 all: $(TARGET)
22
23 # 链接目标可执行文件
24 $(TARGET): $(OBJS)
25      $(CXX) -o $@ $^ $(LDFLAGS)
26
27 # 编译源文件为目标文件
28 $(OBJDIR)/%.o: $(SRCDIR)/%.cpp
29      $(CXX) $(CXXFLAGS) -c -o $@ $<
30
31 # 清理生成的文件
32 clean:
33      rm -f $(TARGET) $(OBJS)
```

## 5 实验结果

### 5.1 展示前准备

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，对端口的状态无法进行判断。因此本次实验的运行平台和测试平台都是 Linux 平台。

如表2所示，本次实验采用的源主机为 Ubuntu 20.04.6 LTS amd64, IPv4 地址为 192.168.126.128。目标主机为 CentOS Linux release 7.9.2009 (Core)，这里主要展示其 IPv4 地址为 10.136.28.34 时的结果。

现在首先对这两个平台进行状态检查，包括检查防火墙的过滤功能，保证端口扫描程序能够正常的接收各种响应数据包。

#### 5.1.1 源主机状态检查

在源主机（192.168.126.128）中通过命令：

---

```
1 service iptables stop
2 sudo systemctl status firewalld
```

---

检查防火墙状态。检查结果为：

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024$ service iptables stop
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-units ===
Authentication is required to stop 'iptables.service'.
Authenticating as: erwinzhou, (erwinzhou)
Password:
==== AUTHENTICATION COMPLETE ===
Failed to stop iptables.service: Unit iptables.service not loaded.
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024$ sudo systemctl status firewalld
[sudo] password for erwinzhou:
Unit firewalld.service could not be found.
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024$ █
```

图 5.7: Ubuntu (192.168.126.128) 的防火墙以及屏蔽工具状态

如图5.7所示，可以看到主机 192.168.126.128 压根没有防火墙，因此也不存在阻挡的问题。

#### 5.1.2 目标主机状态检查

通过命令：

---

```
1 firewall-cmd --state
2 ss -uln
```

---

检查防火墙和端口状态，检查结果为：

```
[hadoop@hadoop101 ~]$ firewall-cmd --state
not running
[hadoop@hadoop101 ~]$ ss -uln
State      Recv-Q Send-Q Local Address:Port                         Peer Address:Port
UNCONN      0      0          *:5353                                *:*
UNCONN      0      0          *:46795                               *:*
UNCONN      0      0          192.168.122.1:53                           *:*
UNCONN      0      0          *%virbr0:67                            *:*
UNCONN      0      0          *:111                                 *:*
UNCONN      0      0          127.0.0.1:323                           *:*
UNCONN      0      0          *:958                                 *:*
UNCONN      0      0          [::]:111                             [::]:*
UNCONN      0      0          [::1]:323                            [::]:*
UNCONN      0      0          [::]:958                             [::]:*
```

图 5.8: Centos(10.136.28.34) 的防火墙以及 UDP 端口打开状态

如图5.8所示，防火墙也 not running 并且 UDP 端口只有 111 是打开的。

## 5.2 实验结果展示

### 5.2.1 显示帮助信息 (-h)

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -h
[sudo] password for erwinzhou:
----- Scanner Help Info -----
Usage: sudo ./Scanner [-Option] <target address> {additional parameters}
Options:
  [-h] --help
  [-p] <target address> {times}
    --ping the <target address> for {times} times
    --if {times} equals 0, it stands for ping until timeout
    --if {times} is empty, it stands for ping once
  [-c] <target address> {begin port} {end port}
    --TCP Connect scan at the <target address> from <begin port> to <end port>
    --{begin port} and {end port} can be empty for manually input
  [-s] --TCP Syn scan
    --TCP Syn scan at the <target address> from <begin port> to <end port>
    --{begin port} and {end port} can be empty for manually input
  [-f] --TCP fin scan
    --TCP Fin scan at the <target address> from <begin port> to <end port>
    --{begin port} and {end port} can be empty for manually input
  [-u] --UDP scan
    --UDP scan at the <target address> from <begin port> to <end port>
    --{begin port} and {end port} can be empty for manually input
-----
```

图 5.9: 显示帮助信息 (-h) 演示

如图5.9所示，可以看到通过在 Scanner 程序后通过-h 选项，可以显示非常详细的帮助信息。包括程序使用的具体 Usage 格式和 Options 每个选项的含义解释。这样的帮助信息对于初次接触该程序的用户来说非常有用，可以快速了解程序的功能和使用方法。

### 5.2.2 ping 连通性测试 (-p)

首先测试 ping 连通性探测 (-p) 的正确性，这里使用一些比较常见的网址如 www.baidu.com((110.242.68.66) scanme.nmap.org(45.33.32.156) 等。

这里为了方便对比，首先使用 Linux 的标准 ping 程序进行测试，结果如下：

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ ping baidu.com
PING baidu.com (110.242.68.66) 56(84) bytes of data.
64 bytes from 110.242.68.66 (110.242.68.66): icmp_seq=1 ttl=128 time=10.2 ms
64 bytes from 110.242.68.66 (110.242.68.66): icmp_seq=2 ttl=128 time=11.3 ms
64 bytes from 110.242.68.66 (110.242.68.66): icmp_seq=3 ttl=128 time=10.7 ms
64 bytes from 110.242.68.66 (110.242.68.66): icmp_seq=4 ttl=128 time=10.5 ms
64 bytes from 110.242.68.66 (110.242.68.66): icmp_seq=5 ttl=128 time=10.7 ms
64 bytes from 110.242.68.66 (110.242.68.66): icmp_seq=6 ttl=128 time=10.9 ms
64 bytes from 110.242.68.66 (110.242.68.66): icmp_seq=7 ttl=128 time=10.9 ms
64 bytes from 110.242.68.66 (110.242.68.66): icmp_seq=8 ttl=128 time=10.9 ms
```

图 5.10: 标准 ping 程序扫描 www.baidu.com

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ ping scanme.nmap.org
PING scanme.nmap.org (45.33.32.156) 56(84) bytes of data.
64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=1 ttl=128 time=157 ms
64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=2 ttl=128 time=157 ms
64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=3 ttl=128 time=157 ms
64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=4 ttl=128 time=157 ms
64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=5 ttl=128 time=157 ms
64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=6 ttl=128 time=157 ms
64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=7 ttl=128 time=158 ms
64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=8 ttl=128 time=158 ms
64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=9 ttl=128 time=157 ms
64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=10 ttl=128 time=156 ms
64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=11 ttl=128 time=156 ms
64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=12 ttl=128 time=157 ms
```

图 5.11: 标准 ping 程序扫描 scanme.nmap.org

如图5.10和5.11所示，可以看到标准的 ping 程序能够显示最开始目标地址，总计发送的比特数，接收到的比特数，`icmp_seq`，Time to live 字段 `ttl`，发送到接收到回复总时间 `time`。

接下来测试我们的 ping 程序即 (`-p`) 选项，还是主要以 `www.baidu.com(39.156.66.10)` 和 `scanme.nmap.org(45.33.32.156)` 为例，分别测试一下功能：

1. 指定探测次数 `time`: 当指定 `time` 为非 0 的正整数时，会对指定地址进行指定次数（以 5 次为例）的扫描，通过命令：

---

```
1 sudo ./Scanner -p 110.242.68.66 5
2 sudo ./Scanner -p 45.33.32.156 5
```

---

结果如下：

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -p 110.242.68.66 5
----- Ping Target IP Address -----
PING 110.242.68.66 (110.242.68.66) 56(84) bytes of data.
104 bytes from 110.242.68.66: icmp_seq=1 ttl=128 time=10.209 ms
104 bytes from 110.242.68.66: icmp_seq=2 ttl=128 time=11.257 ms
104 bytes from 110.242.68.66: icmp_seq=3 ttl=128 time=10.152 ms
104 bytes from 110.242.68.66: icmp_seq=4 ttl=128 time=10.316 ms
104 bytes from 110.242.68.66: icmp_seq=5 ttl=128 time=10.568 ms
[INFO] Ping the target IP address 110.242.68.66 successfully
```

---

图 5.12: ping 程序指定次数探测 www.baidu.com(39.156.66.10)

```
● (base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -p 45.33.32.156 5
----- Ping Target IP Address -----
PING 45.33.32.156 (45.33.32.156) 56(84) bytes of data.
104 bytes from 45.33.32.156: icmp_seq=1 ttl=128 time=155.882 ms
104 bytes from 45.33.32.156: icmp_seq=2 ttl=128 time=156.252 ms
104 bytes from 45.33.32.156: icmp_seq=3 ttl=128 time=156.278 ms
104 bytes from 45.33.32.156: icmp_seq=4 ttl=128 time=157.783 ms
104 bytes from 45.33.32.156: icmp_seq=5 ttl=128 time=156.478 ms
[INFO] Ping the target IP address 45.33.32.156 successfully
```

图 5.13: ping 程序指定次数探测 scanme.nmap.org(45.33.32.156)

如图5.12和5.13所示，可以看到通过程序 Scanner 指定次数（5 次）分别测试

www.baidu.com(39.156.66.10) 和 scanme.nmap.org(45.33.32.156) 连通性都成功了，并且能够清晰地显示和标准 ping 程序一样的日志，即最开始目标地址，总计发送的比特数，接收到的比特数，icmp\_seq，Time to live 字段 ttl，发送到接收到回复总时间 time。证明了 ping 程序指定次数探测功能实现的成功。

- 缺省探测次数 time：接下来以 www.baidu.com(39.156.66.10) 为例，测试缺省探测次数 time 的结果，通过命令：

```
1 sudo ./Scanner -p 110.242.68.66
```

结果如下：

```
● (base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -p 110.242.68.66
[INFO] Using the default ping times 1 for the target IP address: 110.242.68.66
----- Ping Target IP Address -----
PING 110.242.68.66 (110.242.68.66) 56(84) bytes of data.
104 bytes from 110.242.68.66: icmp_seq=1 ttl=128 time=10.516 ms
[INFO] Ping the target IP address 110.242.68.66 successfully
```

图 5.14: ping 程序缺省次数探测 www.baidu.com(39.156.66.10)

如图5.14所示，可以看到在缺省探测次数的情况下，会默认使用 time=1，依然能够探测成功。证明了 ping 程序缺省次数探测功能实现的成功。

- 持续探测 time=0：当指定 ping 程序 time=0 时，会持续探测直到超时，还是以 www.baidu.com(39.156.66.10) 为例，通过命令：

```
1 sudo ./Scanner -p 110.242.68.66 0
```

结果如下：

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -p 110.242.68.66 0
[INFO] Ping the target IP address: 110.242.68.66 until timeout
----- Ping Target IP Address -----
PING 110.242.68.66 (110.242.68.66) 56(84) bytes of data.
104 bytes from 110.242.68.66: icmp_seq=1 ttl=128 time=11.727 ms
104 bytes from 110.242.68.66: icmp_seq=2 ttl=128 time=10.314 ms
104 bytes from 110.242.68.66: icmp_seq=3 ttl=128 time=10.051 ms
104 bytes from 110.242.68.66: icmp_seq=4 ttl=128 time=10.558 ms
104 bytes from 110.242.68.66: icmp_seq=5 ttl=128 time=10.712 ms
104 bytes from 110.242.68.66: icmp_seq=6 ttl=128 time=10.649 ms
104 bytes from 110.242.68.66: icmp_seq=7 ttl=128 time=10.452 ms
104 bytes from 110.242.68.66: icmp_seq=8 ttl=128 time=10.404 ms
104 bytes from 110.242.68.66: icmp_seq=9 ttl=128 time=10.524 ms
104 bytes from 110.242.68.66: icmp_seq=10 ttl=128 time=10.361 ms
104 bytes from 110.242.68.66: icmp_seq=11 ttl=128 time=10.55 ms
104 bytes from 110.242.68.66: icmp_seq=12 ttl=128 time=10.264 ms
104 bytes from 110.242.68.66: icmp_seq=13 ttl=128 time=10.305 ms
```

图 5.15: ping 程序持续探测 www.baidu.com(39.156.66.10)

如图5.15所示，可以看到在指定 `time=0` 后，输出日志信息 `until timeout`，而后持续的探测，直到用户终止。依然能够探测成功。证明了 ping 程序持续探测功能实现的成功。

### 5.2.3 TCP Connect 扫描 (-c)

上面实验结果已经证明了端口扫描器 `ping` 程序实现的正确性。现在我们将其应用到一个本地的虚拟机 Linux 平台上，即源主机 192.168.126.128 向目的主机 10.136.28.34 发起扫描，并使用 **Wireshark 在源主机 192.168.126.128 上抓包**。抓包所使用的 Wireshark 规则为：

---

```
1 (ip.src == 10.136.28.34 and ip.dst == 192.168.126.128) or (ip.src ==
→ 192.168.126.128 and ip.dst == 10.136.28.34)
```

---

抓包结果均可以在 `pkgs` 目录下找到。首先测试的是 TCP Connect 扫描，以 TCP 端口 21-23 为例，分别测试以下功能：

#### 1. 显式给定起止端口

首先显式给定端口 21-23，通过命令对 10.136.28.34 的 21, 22 和 23 三个 TCP 端口发起扫描：

---

```
1 sudo ./Scanner -c 10.136.28.34 21 23
```

---

扫描结果为：

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -c 10.136.28.34 21 23
----- TCP Connect Scan -----
----- Ping Target IP Address -----
PING 10.136.28.34 (10.136.28.34) 56(84) bytes of data.
104 bytes from 10.136.28.34: icmp_seq=1 ttl=128 time=1.56 ms
104 bytes from 10.136.28.34: icmp_seq=2 ttl=128 time=1.051 ms
104 bytes from 10.136.28.34: icmp_seq=3 ttl=128 time=1.004 ms
[INFO] Ping the target IP address 10.136.28.34 successfully
----- [INFO] TCP Connect Scan Host 10.136.28.34 port 21~23...
----- [INFO] Begin TCP connect scan...
[INFO] Host: 10.136.28.34 Port: 21 closed or filtered!
[INFO] Host: 10.136.28.34 Port: 22 open!
[INFO] Host: 10.136.28.34 Port: 23 closed or filtered!
[INFO] TCP Connect Scan Host 10.136.28.34 port 21~23 successfully
```

图 5.16: TCP Connect Scan 显式给定起止端口扫描结果

如图5.16所示，首先启动了 ping 连通性测试。从 ping 连通性测试可以看到，3 次 ping 程序后，源主机 192.168.126.128 可以连通目的主机 10.136.28.34。而后开始 TCP Connect 扫描。能清楚地看到结果为 22 端口开放，21 和 23 端口关闭或被过滤。

打开 Wireshark 查看抓包结果：

No.	Time	Source	Destination	Protocol	Length	Info
492 34. 692449178	192.168.126.128	10.136.28.34		ICMP	118	Echo (ping) request id=0x0050, seq=1/256, ttl=64 (reply in 493)
493 34. 694026819	10.136.28.34	192.168.126.128		ICMP	118	Echo (ping) reply id=0x0050, seq=1/256, ttl=128 (request in 492)
529 35. 694504590	192.168.126.128	10.136.28.34		ICMP	118	Echo (ping) request id=0x0050, seq=2/512, ttl=64 (reply in 530)
530 35. 695574116	10.136.28.34	192.168.126.128		ICMP	118	Echo (ping) reply id=0x0050, seq=2/512, ttl=128 (request in 529)
552 36. 6996187476	192.168.126.128	10.136.28.34		ICMP	118	Echo (ping) request id=0x0050, seq=3/768, ttl=64 (reply in 553)
553 36. 6997214778	10.136.28.34	192.168.126.128		ICMP	118	Echo (ping) reply id=0x0050, seq=3/768, ttl=128 (request in 552)
554 36. 6997874163	192.168.126.128	10.136.28.34		TCP	74	48610 - 21 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2258690220 TSeср=0 WS=1
555 36. 6997990051	192.168.126.128	10.136.28.34		TCP	74	56484 - 22 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2258690220 TSeср=0 WS=1
556 36. 6997992446	192.168.126.128	10.136.28.34		TCP	74	54454 - 23 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2258690220 TSeср=0 WS=1
557 36. 6999252920	10.136.28.34	192.168.126.128		TCP	60	22 - 56484 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
558 36. 6999314616	192.168.126.128	10.136.28.34		TCP	54	56484 - 22 [ACK] Seq=1 Ack=1 Win=64240 Len=0
559 36. 6999496900	192.168.126.128	10.136.28.34		TCP	54	56484 - 22 [FIN, ACK] Seq=1 Ack=1 Win=64240 Len=0
560 36. 6999867684	10.136.28.34	192.168.126.128		TCP	60	22 - 56484 [ACK] Seq=1 Ack=2 Win=64239 Len=0
562 36. 107065538	10.136.28.34	192.168.126.128		SSH	75	Server: Protocol (SSH-2_0_OpenSSH_7.4)
563 36. 107095413	192.168.126.128	10.136.28.34		TCP	54	56484 - 22 [RST] Seq=2 Win=0 Len=0
571 37. 109564899	192.168.126.128	10.136.28.34		TCP	74	[TCP Retransmission] [TCP Port numbers reused] 54454 - 23 [SYN] Seq=0 Win=64240 Len=0 MSS
572 37. 110227961	192.168.126.128	10.136.28.34		TCP	74	[TCP Retransmission] [TCP Port numbers reused] 48610 - 21 [SYN] Seq=0 Win=64240 Len=0 MSS
606 38. 133505811	192.168.126.128	10.136.28.34		TCP	74	[TCP Retransmission] [TCP Port numbers reused] 48610 - 21 [SYN] Seq=0 Win=64240 Len=0 MSS
607 38. 133616199	192.168.126.128	10.136.28.34		TCP	74	[TCP Retransmission] [TCP Port numbers reused] 54454 - 23 [SYN] Seq=0 Win=64240 Len=0 MSS
608 38. 138932134	10.136.28.34	192.168.126.128		TCP	60	21 - 48610 [RST, ACK] Seq=1 Ack=1 Win=64240 Len=0
609 38. 138932585	10.136.28.34	192.168.126.128		TCP	60	23 - 54454 [RST, ACK] Seq=1 Ack=1 Win=64240 Len=0

图 5.17: Wireshark 抓包结果-TCP Connect Scan 显式给定起止端口

如图5.17所示，可以看到：

- 最开始连续的三次 ping Echo Request，以及对应的三次 ping Echo Reply，进一步证明了连通性测试的正确性。
- 中间由本地主机向目的主机连续发送了三次 SYN 数据包，其中目的主机在 22 端口回复了 SYN|ACK，而后二者开始 TCP 建立连接的三次握手，证明了 22 端口的开放测试结果的正确性。
- 最后看到 21 和 23 端口源主机发送了 RST|ACK，证明了两个端口不是开启的。但需要进一步判断是被过滤了还是关闭了。

## 2. 手动输入起止端口

接下来缺省起止端口，通过命令：

```
1 sudo ./Scanner -c 10.136.28.34
```

结果如下：

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -c 10.136.28.34
----- TCP Connect Scan -----
[INFO] Please input the range of port(0-65535)...
Begin port: 21
End port: 23
----- Ping Target IP Address -----
PING 10.136.28.34 (10.136.28.34) 56(84) bytes of data.
104 bytes from 10.136.28.34: icmp_seq=1 ttl=128 time=0.85 ms
104 bytes from 10.136.28.34: icmp_seq=2 ttl=128 time=1.072 ms
104 bytes from 10.136.28.34: icmp_seq=3 ttl=128 time=0.77 ms
[INFO] Ping the target IP address 10.136.28.34 successfully
-----
[INFO] TCP Connect Scan Host 10.136.28.34 port 21~23...
[INFO] Begin TCP connect scan...
[INFO] Host: 10.136.28.34 Port: 21 closed or filtered!
[INFO] Host: 10.136.28.34 Port: 22 open!
[INFO] Host: 10.136.28.34 Port: 23 closed or filtered!
[INFO] TCP Connect Scan Host 10.136.28.34 port 21~23 successfully
-----
```

图 5.18: TCP Connect Scan 手动输入起止端口扫描结果

如图5.18所示，当缺省参数时，会提示手动输入起始和终止端口，结果和显式给定一致，证明了交互输入功能实现的正确性。

### 3. 与 Nmap 扫描结果对比

接下来为了进一步验证程序正确性，或者发现并分析潜在的不足之处，通过 Nmap 同样使用 **TCP Connect 扫描** 10.136.28.34 的 TCP 端口，通过命令：

---

```
1 sudo nmap -sT 10.136.38.24
```

---

探测结果如下图所示：

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo nmap -sT 10.136.28.34
Starting Nmap 7.80 ( https://nmap.org ) at 2024-06-03 19:42 CST
Nmap scan report for 10.136.28.34
Host is up (0.0011s latency).
Not shown: 998 filtered ports
PORT      STATE SERVICE
22/tcp    open  ssh
111/tcp   open  rpcbind
```

Nmap done: 1 IP address (1 host up) scanned in 4.94 seconds  
 ○ (base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin\$ █

图 5.19: Nmap TCP Connect 扫描结果

如图5.24所示，可以看到 Nmap 扫描结果为除了 22 和 111 两个 TCP 端口打开，其余均被过滤。虽然 10.136.28.34 的防火墙是关闭的，但是仍然显示了被过滤的结果。推测着点可能和 Nmap 的实现机理有关。不过 22 端口开放的结果和端口扫描器的 TCP Connect 扫描结果一致，进一步验证了实现正确性。

而后通过 Wireshark 抓取 Nmap TCP Connect 的数据包，抓包分析：

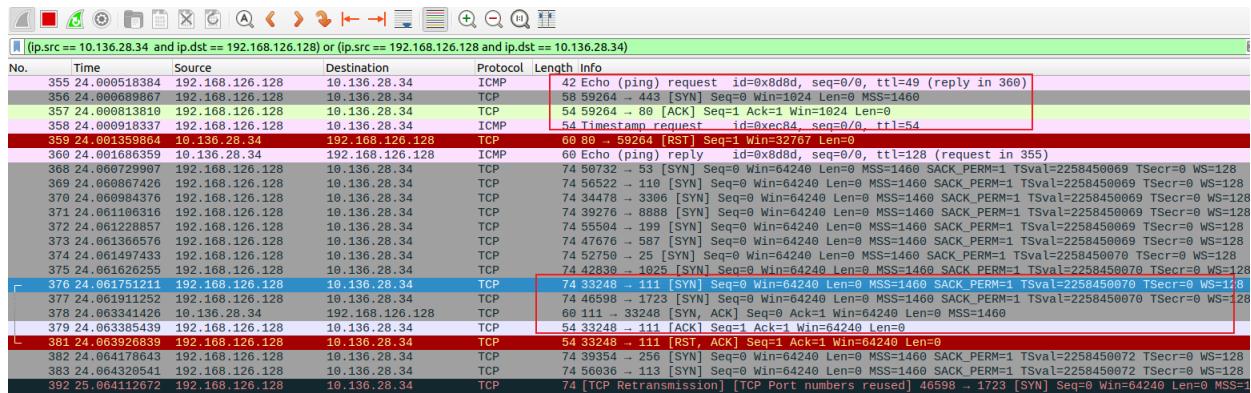


图 5.20: WireShark 抓包结果-Nmap TCP Connect

如图5.20所示，Nmap 首先会通过简单的 ping 探测目标主机的可达性，对于 111 端口来说，可以看到目标主机的 111 端口发送的 SYN|ACK 数据包回复，之后也开启了三次握手。这些都和端口扫描器的一致，验证了端口扫描器的实现能够一定程度上对 Nmap 进行靠近，其有效性得到了有力验证。

#### 5.2.4 TCP Syn 扫描 (-s)

接下来测试的是 TCP Syn 扫描，仍然以 10.136.28.34 的 TCP 端口 21-23 为例，继续测试以下功能：

##### 1. 显式给定起止端口

通过命令：

```
1 sudo ./Scanner -s 10.136.28.34 21 23
```

扫描结果为：

```
● (base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -s 10.136.28.34 21 23
----- TCP Syn Scan -----
----- Ping Target IP Address -----
PING 10.136.28.34 (10.136.28.34) 56(84) bytes of data.
104 bytes from 10.136.28.34: icmp_seq=1 ttl=128 time=0.972 ms
104 bytes from 10.136.28.34: icmp_seq=2 ttl=128 time=0.997 ms
104 bytes from 10.136.28.34: icmp_seq=3 ttl=128 time=1.035 ms
[INFO] Ping the target IP address 10.136.28.34 successfully
-----
[INFO] TCP Syn Scan Host 10.136.28.34 port 21~23...
[INFO] Begin TCP syn scan...
[INFO] Host: 10.136.28.34 Port: 21 filtered!
[INFO] Host: 10.136.28.34 Port: 22 open!
[INFO] Host: 10.136.28.34 Port: 23 filtered!
[INFO] TCP Syn Scan Host 10.136.28.34 port 21~23 successfully
```

图 5.21: TCP Syn Scan 显式给定起止端口扫描结果

如图5.21所示，在 ping 通了目标主机后，扫描端口显式结果为 22 端口开放，21 和 23 端口被过滤。这一点和 Connect 扫描结果相同。打开 WireShark 查看抓包结果：

(ip.src == 10.136.28.34 and ip.dst == 192.168.126.128) or (ip.src == 192.168.126.128 and ip.dst == 10.136.28.34)						
No.	Time	Source	Destination	Protocol	Length	Info
365 8. 9.267687369	192.168.126.128	10.136.28.34	192.168.126.128	ICMP	118	Echo (ping) request id=0x0050, seq=1/256, ttl=64 (reply in 366)
366 8. 9.27687369	10.136.28.34	192.168.126.128	10.136.28.34	ICMP	118	Echo (ping) reply id=0x0050, seq=1/256, ttl=128 (request in 365)
388 9. 9.28770558	192.168.126.128	10.136.28.34	192.168.126.128	ICMP	118	Echo (ping) request id=0x0050, seq=2/512, ttl=64 (reply in 389)
389 9. 9.29820163	10.136.28.34	192.168.126.128	10.136.28.34	ICMP	118	Echo (ping) reply id=0x0050, seq=2/512, ttl=128 (request in 388)
398 10. 9.030302446	192.168.126.128	10.136.28.34	192.168.126.128	ICMP	118	Echo (ping) request id=0x0050, seq=3/768, ttl=64 (reply in 399)
399 10. 9.031390177	10.136.28.34	192.168.126.128	10.136.28.34	ICMP	118	Echo (ping) reply id=0x0050, seq=3/768, ttl=128 (request in 398)
400 10. 9.032360057	192.168.126.128	10.136.28.34	192.168.126.128	TCP	54	80 - 21 [SYN] Seq=0 Win=65535 Len=0
401 10. 9.032777189	192.168.126.128	10.136.28.34	192.168.126.128	TCP	54	80 - 22 [SYN] Seq=0 Win=65535 Len=0
402 10. 9.033521942	192.168.126.128	10.136.28.34	192.168.126.128	TCP	54	80 - 23 [SYN] Seq=0 Win=65535 Len=0
416 12. 9.825066159	10.136.28.34	192.168.126.128	10.136.28.34	TCP	60	22 - 80 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
417 12. 9.82542417	192.168.126.128	10.136.28.34	192.168.126.128	TCP	54	80 - 22 [RST] Seq=1 Win=0 Len=0
442 15. 0.030094787	10.136.28.34	192.168.126.128	10.136.28.34	SSH	75	Server: Protocol (SSH-2.0-OpenSSH_7.4)
443 15. 0.030135764	192.168.126.128	10.136.28.34	192.168.126.128	TCP	54	80 - 22 [RST] Seq=1 Win=0 Len=0

图 5.22: Wireshark 抓包结果-TCP Syn Scan 显式给定起止端口

如图5.22所示，可以看到中间的源主机分别向目标主机的 21, 22 和 23 端口都发送了 SYN 数据包，其中只有 22 端口回复了 SYN|ACK 消息，验证了端口的开启。其它两个端口均没有消息，大概率被过滤掉了。

## 2. 手动输入起止端口

接下来验证手动输入起止端口：

```
1 sudo ./Scanner -s 10.136.28.34
```

结果如下：

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -s 10.136.28.34
----- TCP Syn Scan -----
[INFO] Please input the range of port(0-65535)...
Begin port: 21
End port: 23
----- Ping Target IP Address -----
PING 10.136.28.34 (10.136.28.34) 56(84) bytes of data.
104 bytes from 10.136.28.34: icmp_seq=1 ttl=128 time=0.915 ms
104 bytes from 10.136.28.34: icmp_seq=2 ttl=128 time=0.925 ms
104 bytes from 10.136.28.34: icmp_seq=3 ttl=128 time=2.25 ms
[INFO] Ping the target IP address 10.136.28.34 successfully
-----
[INFO] TCP Syn Scan Host 10.136.28.34 port 21~23...
[INFO] Begin TCP syn scan...
[INFO] Host: 10.136.28.34 Port: 21 filtered!
[INFO] Host: 10.136.28.34 Port: 22 open!
[INFO] Host: 10.136.28.34 Port: 23 filtered!
[INFO] TCP Syn Scan Host 10.136.28.34 port 21~23 successfully
```

图 5.23: TCP Syn Scan 手动输入起止端口扫描结果

如图5.23所示，当缺省参数时，会提示手动输入起始和终止端口，结果和显式给定一致，证明了交互输入功能实现的正确性。

## 3. 与 Nmap 扫描结果对比

最后依然使用 Nmap 扫描后将结果对比，同样使用 TCP Syn 扫描 10.136.28.34 的 TCP 端口，分别扫描 21-23 和全部端口，通过命令：

```
1 sudo nmap -p 21-23 10.136.28.34
2 sudo nmap -sS 10.136.38.24
```

探测结果如下图所示：

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo nmap -p 21-23 10.136.28.34
Starting Nmap 7.80 ( https://nmap.org ) at 2024-06-03 19:52 CST
Nmap scan report for 10.136.28.34
Host is up (0.0011s latency).

PORT      STATE    SERVICE
21/tcp     filtered  ftp
22/tcp     open      ssh
23/tcp     filtered  telnet

Nmap done: 1 IP address (1 host up) scanned in 1.33 seconds
```

图 5.24: Nmap TCP Syn 扫描结果 (21-23)

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo nmap -sS 10.136.28.34
Starting Nmap 7.80 ( https://nmap.org ) at 2024-06-03 20:12 CST
Nmap scan report for 10.136.28.34
Host is up (2.0s latency).
Not shown: 997 closed ports
PORT      STATE    SERVICE
22/tcp     open      ssh
111/tcp    open      rpcbind
514/tcp    filtered shell

Nmap done: 1 IP address (1 host up) scanned in 206.57 seconds
```

图 5.25: Nmap TCP Syn 扫描结果全部端口

如图5.25所示，可以看到 Nmap 扫描结果为除了 22 和 111 两个 TCP 端口打开，其余均被过滤。虽然 10.136.28.34 的防火墙是关闭的，但是仍然显示了被过滤的结果。推测这可能和 Nmap 的实现机理有关。不过 22 端口开放的结果和端口扫描器的 TCP Connect 扫描结果一致，进一步验证了实现正确性。

而后通过 Wireshark 抓取 Nmap Syn Connect 在 21-23 端口扫描的数据包，抓包分析：

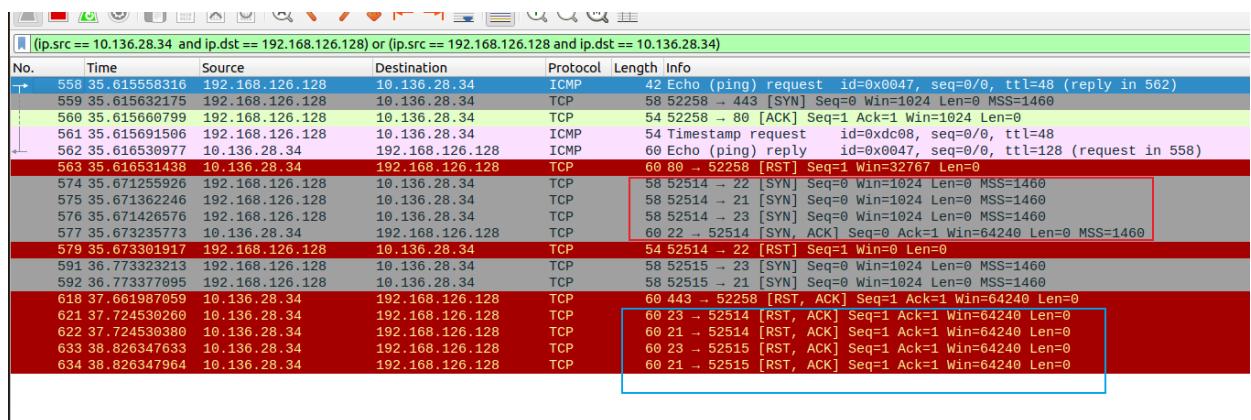


图 5.26: Wireshark 抓包结果-Nmap TCP Syn 对 21-23 端口

如图5.26所示，Nmap 首先会通过简单的 ping 探测目标主机的可达性，而后还是由源本地向目标主机的 21-23 端口发送 SYN 数据包，可以看到只有 21 端口的目的主机向其回复了 SYN|ACK 数据包，而 23 和 21 端口在下面也可以看到在 Nmap 进一步向这两个端口发送了 SYN 数据包后，回复了 RST|ACK 数据包，代表了两个端口时关闭状态的。而我们之前的 SYN 扫描得出的结论是 21 和 23 是被过滤的，与 Nmap 结果有些许区别，我推测原因是在于 Nmap 在第一次没有收

到回复数据包后进一步发送了数据包而后才得到了回复 RST|ACK，因此正是因为 Nmap 这种反  
复确认才下结论的方式才使得它的测试十分准确。

### 5.2.5 TCP Fin 扫描 (-f)

接下来测试的是 TCP Fin 扫描，仍然以 10.136.28.34 的 TCP 端口 21-23 为例，继续测试以下功能：

#### 1. 显式给定起止端口

通过命令：

```
1 sudo ./Scanner -f 10.136.28.34 21 23
```

扫描结果为：

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -f 10.136.28.34 21 23
----- TCP Fin Scan -----
----- Ping Target IP Address -----
PING 10.136.28.34 (10.136.28.34) 56(84) bytes of data.
104 bytes from 10.136.28.34: icmp_seq=1 ttl=128 time=1.273 ms
104 bytes from 10.136.28.34: icmp_seq=2 ttl=128 time=0.956 ms
104 bytes from 10.136.28.34: icmp_seq=3 ttl=128 time=1.09 ms
[INFO] Ping the target IP address 10.136.28.34 successfully

[INFO] TCP Fin Scan Host 10.136.28.34 port 21~23...
[INFO] Begin TCP fin scan...
[INFO] Host: 10.136.28.34 Port: 21 open!
[INFO] Host: 10.136.28.34 Port: 22 open!
[INFO] Host: 10.136.28.34 Port: 23 open!
[INFO] TCP Fin Scan Host 10.136.28.34 port 21~23 successfully
```

图 5.27: TCP Fin Scan 显式给定起止端口扫描结果

如图5.27所示，在 ping 通了目标主机后，扫描端口显式结果为 21-23 端口全部开放，这一段和之前的 Connect 以及 Syn 扫描结果均不相同。

打开 Wireshark 查看抓包结果：

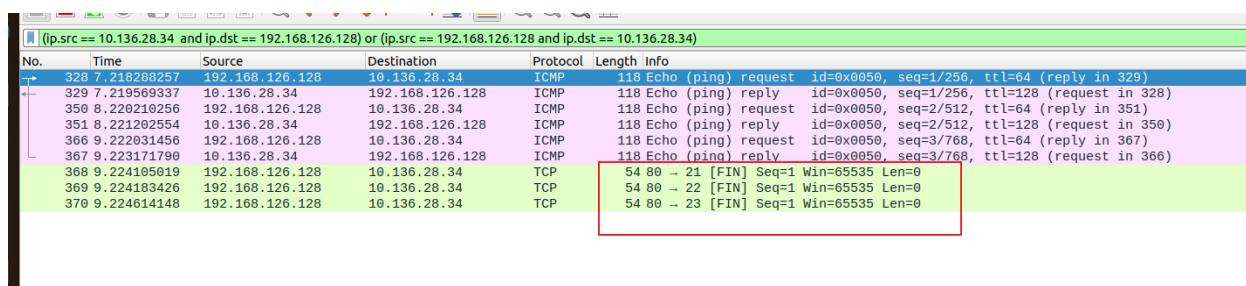


图 5.28: Wireshark 抓包结果-TCP Fin Scan 显式给定起止端口

如图5.28所示，可以看到中间的源主机分别向目标主机的 21, 22 和 23 端口都发送了 Fin 数据包，但是之后没有得到任何回复，按照 Fin 扫描的规则，由于目标主机没有任何响应，并且之前的 ping 程序成功收取到回复证明主机存在，因此可以得到结论：目标主机在监听这三个端口并且端口开放。这和我们的显示结果相同。

由于没有收到任何 RST 响应，无法得出端口关闭的结论。另外值得一提的是，我由于是对 Centos(Linux) 系统进行的扫描，因此也不会存在 Windows 系统导致判断出现错误的情况。

## 2. 手动输入起止端口

接下来验证手动输入起止端口：

---

```
1 sudo ./Scanner -f 10.136.28.34
```

---

结果如下：

```
● (base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -f 10.136.28.34
----- TCP Fin Scan -----
[INFO] Please input the range of port(0-65535)...
Begin port: 21
End port: 23
----- Ping Target IP Address -----
PING 10.136.28.34 (10.136.28.34) 56(84) bytes of data.
104 bytes from 10.136.28.34: icmp_seq=1 ttl=128 time=0.899 ms
104 bytes from 10.136.28.34: icmp_seq=2 ttl=128 time=0.739 ms
104 bytes from 10.136.28.34: icmp_seq=3 ttl=128 time=0.821 ms
[INFO] Ping the target IP address 10.136.28.34 successfully
-----
[INFO] TCP Fin Scan Host 10.136.28.34 port 21~23...
[INFO] Begin TCP fin scan...
[INFO] Host: 10.136.28.34 Port: 21 open!
[INFO] Host: 10.136.28.34 Port: 22 open!
[INFO] Host: 10.136.28.34 Port: 23 open!
[INFO] TCP Fin Scan Host 10.136.28.34 port 21~23 successfully
```

图 5.29: TCP Fin Scan 手动输入起止端口扫描结果

如图5.29所示，当缺省参数时，会提示手动输入起始和终止端口，结果和显式给定一致，证明了交互输入功能实现的正确性。

## 3. 与 Nmap 扫描结果对比

最后依然使用 Nmap 扫描后将结果对比，同样使用 TCP Fin 扫描 10.136.28.34 的全部 TCP 端口，通过命令：

---

```
1 sudo nmap -sF 10.136.38.24
```

---

探测结果如下图所示：

```
● (base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo nmap -p 21-23 10.136.28.34
Starting Nmap 7.80 ( https://nmap.org ) at 2024-06-03 19:52 CST
Nmap scan report for 10.136.28.34
Host is up (0.0011s latency).

PORT      STATE      SERVICE
21/tcp    filtered  ftp
22/tcp    open       ssh
23/tcp    filtered  telnet

Nmap done: 1 IP address (1 host up) scanned in 1.33 seconds
```

图 5.30: Nmap TCP Fin 扫描结果全部端口

如图5.30所示，可以看到 Nmap 扫描结果为所有 10.136.28.34 端口均为打开或者过滤的，这一点和我们的端口扫描器结果一致，验证了端口扫描器结果的正确性。但是 Nmap 自己的扫描结果和之前的 SYN 以及 Connect 扫描不同，这一点我会在特殊结果分析中提到。

而后通过 Wireshark 抓取 Nmap TCP Fin 在所有端口扫描的数据包，抓包分析：

No.	Time	Source	Destination	Protocol	Length	Info
153	8.078285589	192.168.126.128	10.136.28.34	TCP	54	58995 - 250 [FIN] Seq=1 Win=1024 Len=0
154	8.078356322	192.168.126.128	10.136.28.34	TCP	54	58995 - 1723 [FIN] Seq=1 Win=1024 Len=0
155	8.078386398	192.168.126.128	10.136.28.34	TCP	54	58995 - 199 [FIN] Seq=1 Win=1024 Len=0
156	8.078419570	192.168.126.128	10.136.28.34	TCP	54	58995 - 8888 [FIN] Seq=1 Win=1024 Len=0
157	8.078451891	192.168.126.128	10.136.28.34	TCP	54	58995 - 143 [FIN] Seq=1 Win=1024 Len=0
158	8.078521862	192.168.126.128	10.136.28.34	TCP	54	58995 - 113 [FIN] Seq=1 Win=1024 Len=0
159	8.078558311	192.168.126.128	10.136.28.34	TCP	54	58995 - 993 [FIN] Seq=1 Win=1024 Len=0
160	8.078596562	192.168.126.128	10.136.28.34	TCP	54	58995 - 1025 [FIN] Seq=1 Win=1024 Len=0
161	8.078631538	192.168.126.128	10.136.28.34	TCP	54	58995 - 1720 [FIN] Seq=1 Win=1024 Len=0
162	8.078666594	192.168.126.128	10.136.28.34	TCP	54	58995 - 110 [FIN] Seq=1 Win=1024 Len=0
163	8.179424488	192.168.126.128	10.136.28.34	TCP	54	58106 - 80 [ACK] Seq=1 Ack=1 Win=1024 Len=0
164	8.179563528	192.168.126.128	10.136.28.34	TCP	54	58995 - 110 [FIN] Seq=1 Win=1024 Len=0
165	8.179670329	192.168.126.128	10.136.28.34	TCP	54	58995 - 1720 [FIN] Seq=1 Win=1024 Len=0
166	8.179787809	192.168.126.128	10.136.28.34	TCP	54	58996 - 1025 [FIN] Seq=1 Win=1024 Len=0
167	8.179836160	192.168.126.128	10.136.28.34	TCP	54	58996 - 993 [FIN] Seq=1 Win=1024 Len=0
168	8.179843614	10.136.28.34	192.168.126.128	TCP	60	80 - 58106 [RST] Seq=1 Win=32767 Len=0
169	8.179867639	192.168.126.128	10.136.28.34	TCP	54	58996 - 113 [FIN] Seq=1 Win=1024 Len=0
170	8.179893707	192.168.126.128	10.136.28.34	TCP	54	58996 - 143 [FIN] Seq=1 Win=1024 Len=0
171	8.179925527	192.168.126.128	10.136.28.34	TCP	54	58996 - 8888 [FIN] Seq=1 Win=1024 Len=0
172	8.179956756	192.168.126.128	10.136.28.34	TCP	54	58996 - 199 [FIN] Seq=1 Win=1024 Len=0

图 5.31: Wireshark 抓包结果 1-Nmap TCP Fin 对全部端口扫描

首先如图5.31所示，可以看到结果抓包结果中在 80 端口目标主机对源主机回复了 RST 数据包，看似这是端口关闭的标志，实际上仔细利用 Wireshark 工具，从左侧的箭头可以看到，实际上目标主机是在对源主机发送的 ACK 数据包的回复，与端口状态无关，推测可能是与 Nmap 的其他机制有关。

No.	Time	Source	Destination	Protocol	Length	Info
681	8.801866848	192.168.126.128	10.136.28.34	TCP	54	58095 - 1443 [FIN] Seq=1 Win=1024 Len=0
682	8.801898344	192.168.126.128	10.136.28.34	TCP	54	58095 - 8181 [FIN] Seq=1 Win=1024 Len=0
683	8.801927348	192.168.126.128	10.136.28.34	TCP	54	58095 - 2135 [FIN] Seq=1 Win=1024 Len=0
684	8.801956292	192.168.126.128	10.136.28.34	TCP	54	58095 - 9594 [FIN] Seq=1 Win=1024 Len=0
685	8.801988493	192.168.126.128	10.136.28.34	TCP	54	58095 - 3000 [FIN] Seq=1 Win=1024 Len=0
686	8.802034780	192.168.126.128	10.136.28.34	TCP	54	58095 - 4005 [FIN] Seq=1 Win=1024 Len=0
687	8.802071258	192.168.126.128	10.136.28.34	TCP	54	58095 - 2022 [FIN] Seq=1 Win=1024 Len=0
688	8.802097086	192.168.126.128	10.136.28.34	TCP	54	58095 - 10004 [FIN] Seq=1 Win=1024 Len=0
689	8.802123085	192.168.126.128	10.136.28.34	TCP	54	58095 - 8083 [FIN] Seq=1 Win=1024 Len=0
690	8.855561177	10.136.28.34	192.168.126.128	TCP	60	443 - 57839 [RST, ACK] Seq=1 Ack=1 Win=64240 Len=0
691	8.882744754	192.168.126.128	10.136.28.34	TCP	54	58095 - 5902 [FIN] Seq=1 Win=1024 Len=0
692	8.882869698	192.168.126.128	10.136.28.34	TCP	54	58095 - 1055 [FIN] Seq=1 Win=1024 Len=0
693	8.885707691	192.168.126.128	10.136.28.34	TCP	54	58095 - 4002 [FIN] Seq=1 Win=1024 Len=0
694	8.885810323	192.168.126.128	10.136.28.34	TCP	54	58095 - 2910 [FIN] Seq=1 Win=1024 Len=0

图 5.32: Wireshark 抓包结果 2-Nmap TCP Fin 对全部端口扫描

No.	Time	Source	Destination	Protocol	Length	Info
73	6.816293078	192.168.126.128	10.136.28.34	ICMP	42	Echo (ping) request id=0xa1ff, seq=0/0, ttl=16 (reply in 78)
74	6.816397123	192.168.126.128	10.136.28.34	TCP	54	57839 - 443 [SYN] Seq=0 Win=1024 MSS=1460
75	6.816457687	192.168.126.128	10.136.28.34	TCP	54	57839 - 80 [ACK] Seq=1 Ack=1 Win=1024 Len=0
76	6.816521156	192.168.126.128	10.136.28.34	ICMP	54	Timestamp request id=0x8e83, seq=0/0, ttl=38
77	6.817232849	10.136.28.34	192.168.126.128	TCP	60	80 - 57839 [RST] Seq=1 Win=32767 Len=0
78	6.817232849	10.136.28.34	192.168.126.128	ICMP	60	Echo (ping) reply id=0xa1ff, seq=0/0, ttl=128 (request in 73)
90	6.876115483	192.168.126.128	10.136.28.34	TCP	54	58095 - 8080 [FIN] Seq=1 Win=1024 Len=0
91	6.876224066	192.168.126.128	10.136.28.34	TCP	54	58095 - 135 [FIN] Seq=1 Win=1024 Len=0
92	6.876257489	192.168.126.128	10.136.28.34	TCP	54	58095 - 111 [FIN] Seq=1 Win=1024 Len=0
93	6.876290591	192.168.126.128	10.136.28.34	TCP	54	58095 - 3389 [FIN] Seq=1 Win=1024 Len=0

图 5.33: Wireshark 抓包结果 3-Nmap TCP Fin 对全部端口扫描

如图5.32和5.33所示，可以看到同样出现了 443 端口由目标主机向源主机回复的 RST 数据包，看似这也是端口关闭的标志，实际上仔细利用 Wireshark 工具，依然是从左侧的箭头可以看到，这也是对最开始 Nmap 探测主机可达性时候的某个 SYN 数据包的回复，也与 Nmap 其它机制有关。

### 5.2.6 UDP 扫描 (-u)

接下来测试的是 UDP 扫描，仍然以 10.136.28.34 的 UDP 端口 110-112 为例，继续测试以下功能：

#### 1. 显式给定起止端口

通过命令：

```
1 sudo ./Scanner -u 10.136.28.34 110 112
```

扫描结果为：

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -u 10.136.28.34 110 112
----- UDP Scan -----
----- Ping Target IP Address -----
PING 10.136.28.34 (10.136.28.34) 56(84) bytes of data.
104 bytes from 10.136.28.34: icmp_seq=1 ttl=128 time=1.077 ms
104 bytes from 10.136.28.34: icmp_seq=2 ttl=128 time=1.187 ms
104 bytes from 10.136.28.34: icmp_seq=3 ttl=128 time=1.41 ms
[INFO] Ping the target IP address 10.136.28.34 successfully
----- [INFO] UDP Scan Host 10.136.28.34 port 110~112...
----- [INFO] Begin UDP scan...
[INFO] Host: 10.136.28.34 Port: 110 open or filtered!
[INFO] Host: 10.136.28.34 Port: 111 open or filtered!
[INFO] Host: 10.136.28.34 Port: 112 open or filtered!
[INFO] UDP Scan Host 10.136.28.34 port 110~112 successfully
```

图 5.34: UDP Scan 显式给定起止端口扫描结果

如图5.34所示，在 ping 通了目标主机后，扫描端口显式结果为 110-112 三个端口全部开放或者被过滤掉了。

打开 Wireshark 查看抓包结果：

(ip.src == 10.136.28.34 and ip.dst == 192.168.126.128) or (ip.src == 192.168.126.128 and ip.dst == 10.136.28.34)						
No.	Time	Source	Destination	Protocol	Length	Info
614	11.849504101	192.168.126.128	10.136.28.34	ICMP	118	Echo (ping) request id=0x0050, seq=1/256, ttl=64 (reply in 615)
615	11.850582119	10.136.28.34	192.168.126.128	ICMP	118	Echo (ping) reply id=0x0050, seq=1/256, ttl=128 (request in 614)
638	12.851144262	192.168.126.128	10.136.28.34	ICMP	118	Echo (ping) request id=0x0050, seq=2/512, ttl=64 (reply in 639)
639	12.852349162	10.136.28.34	192.168.126.128	ICMP	118	Echo (ping) reply id=0x0050, seq=2/512, ttl=128 (request in 638)
653	13.853295047	192.168.126.128	10.136.28.34	ICMP	118	Echo (ping) request id=0x0050, seq=3/768, ttl=64 (reply in 654)
654	13.854763507	10.136.28.34	192.168.126.128	ICMP	118	Echo (ping) reply id=0x0050, seq=3/768, ttl=128 (request in 653)
655	13.855231324	192.168.126.128	10.136.28.34	UDP	42	80 → 110 Len=0
680	18.855447625	192.168.126.128	10.136.28.34	UDP	42	80 → 111 Len=0
701	23.855644228	192.168.126.128	10.136.28.34	UDP	42	80 → 112 Len=0

图 5.35: Wireshark 抓包结果-UDP Scan 显式给定起止端口

如图5.35所示，可以看到中间的源主机分别向目标主机的 110-112 三个端口都发送了 UDP 数据包，但是之后没有得到任何回复，按照 UDP 扫描的规则，由于目标主机没有任何响应，并且之前的 ping 程序成功收取到回复证明主机存在，因此可以得到结论：目标主机在监听这三个端口并且端口开放。这和我们的显示结果相同。由于没有收到任何 ICMP 响应，无法得出端口关闭的结论。

#### 2. 手动输入起止端口

接下来验证手动输入起止端口：

---

```
1 sudo ./Scanner -u 10.136.28.34
```

---

结果如下：

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -u 10.136.28.34
----- UDP Scan -----
[INFO] Please input the range of port(0-65535)...
Begin port: 110
End port: 112
----- Ping Target IP Address -----
PING 10.136.28.34 (10.136.28.34) 56(84) bytes of data.
104 bytes from 10.136.28.34: icmp_seq=1 ttl=128 time=0.857 ms
104 bytes from 10.136.28.34: icmp_seq=2 ttl=128 time=0.773 ms
104 bytes from 10.136.28.34: icmp_seq=3 ttl=128 time=0.994 ms
[INFO] Ping the target IP address 10.136.28.34 successfully

[INFO] UDP Scan Host 10.136.28.34 port 110~112...
[INFO] Begin UDP scan...
[INFO] Host: 10.136.28.34 Port: 110 open or filtered!
[INFO] Host: 10.136.28.34 Port: 111 open or filtered!
[INFO] Host: 10.136.28.34 Port: 112 open or filtered!
[INFO] UDP Scan Host 10.136.28.34 port 110~112 successfully
```

---

图 5.36: TCP UDP Scan 手动输入起止端口扫描结果

如图5.36所示，当缺省参数时，会提示手动输入起始和终止端口，结果和显式给定一致，证明了交互输入功能实现的正确性。

### 3. 与 Nmap 扫描结果对比

最后依然使用 Nmap 扫描后将结果对比，同样使用 UDP 扫描 10.136.28.34 的全部 UDP 端口，通过命令：

---

```
1 sudo nmap -sU 10.136.38.24
```

---

探测结果如下图所示：

```
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo nmap -sU 10.136.28.34
Starting Nmap 7.80 ( https://nmap.org ) at 2024-06-03 20:17 CST
Nmap scan report for 10.136.28.34
Host is up (0.0013s latency).

Not shown: 999 open|filtered ports
PORT      STATE SERVICE
111/udp    open   rpcbind

Nmap done: 1 IP address (1 host up) scanned in 4.52 seconds
```

图 5.37: Nmap UDP 扫描结果全部端口

如图5.37所示，可以看到 Nmap 扫描结果为端口 111 是确定打开的但是 10.136.28.34 的其它端口均为 open|filtered 即打开或者被过滤均有可能。这与我们的端口扫描器不同，得出了很确定的结论。

而后通过 Wireshark 抓取 Nmap UDP 在所有端口扫描的数据包，抓包分析：

(ip.src == 10.136.28.34 and ip.dst == 192.168.126.128) or (ip.src == 192.168.126.128 and ip.dst == 10.136.28.34)						
No.	Time	Source	Destination	Protocol	Length	Info
2061	8.0.025020343	192.168.126.128	10.136.28.34	UDP	42	50006 - 47981 Len=0
2062	8.0.025129822	192.168.126.128	10.136.28.34	UDP	42	50006 - 1045 Len=0
2063	8.0.0252333466	192.168.126.128	10.136.28.34	UDP	42	50006 - 18985 Len=0
2064	8.0.025357296	192.168.126.128	10.136.28.34	UDP	42	50005 - 775 Len=0
2065	8.0.027825547	192.168.126.128	10.136.28.34	UDP	42	50006 - 5003 Len=0
2066	8.0.028676285	192.168.126.128	10.136.28.34	UDP	42	50006 - 49193 Len=0
2067	8.0.028303428	192.168.126.128	10.136.28.34	UDP	42	50006 - 19605 Len=0
2068	8.0.028570788	192.168.126.128	10.136.28.34	UDP	42	50006 - 657 Len=0
2069	8.0.028682912	192.168.126.128	10.136.28.34	UDP	42	50006 - 21358 Len=0
2070	8.0.028786389	192.168.126.128	10.136.28.34	DTLS	109	Client Hello
2071	8.0.028894826	192.168.126.128	10.136.28.34	UDP	42	50006 - 3296 Len=0
2072	8.0.031662098	192.168.126.128	10.136.28.34	Portmap	82	[RPC retransmission of #229]V104316 proc-0 Call (Reply In 239)
2073	8.0.031785464	192.168.126.128	10.136.28.34	UDP	42	50006 - 65024 Len=0
2074	8.0.031889292	192.168.126.128	10.136.28.34	UDP	42	50006 - 57843 Len=0
2075	8.0.031990314	192.168.126.128	10.136.28.34	UDP	42	50006 - 21525 Len=0
2076	8.0.032695816	192.168.126.128	10.136.28.34	UDP	42	50006 - 17580 Len=0
2077	8.0.032321045	192.168.126.128	10.136.28.34	UDP	42	50006 - 54711 Len=0
2078	8.0.032435173	192.168.126.128	10.136.28.34	UDP	42	50006 - 50788 Len=0
2079	8.0.032538800	192.168.126.128	10.136.28.34	UDP	42	50006 - 17533 Len=0
2080	8.0.033641919	10.136.28.34	192.168.126.128	Portmap	74	[RPC duplicate of #239]V104316 proc-0 Reply (Call In 229)
2081	8.0.033683929	192.168.126.128	10.136.28.34	ICMP	102	Destination unreachable (Port unreachable)
2082	8.0.036541913	192.168.126.128	10.136.28.34	UDP	42	50006 - 33030 Len=0
2083	8.0.036677462	192.168.126.128	10.136.28.34	UDP	42	50006 - 35438 Len=0

图 5.38: WireShark 抓包结果 1-Nmap UDP 对全部端口扫描

首先如图5.38所示，可以看到接收到的 ICMP 数据包，但是是从源主机发向目的主机的，同时再次利用 WireShark 边上的工具线，可以看到如下：

(ip.src == 10.136.28.34 and ip.dst == 192.168.126.128) or (ip.src == 192.168.126.128 and ip.dst == 10.136.28.34)						
No.	Time	Source	Destination	Protocol	Length	Info
214	4.0.097152230	192.168.126.128	10.136.28.34	ICMP	42	Echo (ping) request id=0x924f, seq=0/0, ttl=59 (reply in 218)
215	4.0.097223416	192.168.126.128	10.136.28.34	TCP	58	49749 - 443 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
216	4.0.097251269	192.168.126.128	10.136.28.34	TCP	54	49749 - 80 [ACK] Seq=1 Ack=1 Win=1024 Len=0
217	4.0.097279852	192.168.126.128	10.136.28.34	ICMP	54	Timestamp request id=0x95f0, seq=0/0, ttl=38
218	4.0.098103126	10.136.28.34	192.168.126.128	TCP	60	Echo (ping) reply id=0x924f, seq=0/0, ttl=128 (request in 214)
219	4.0.098103427	10.136.28.34	192.168.126.128	TCP	60	RQ 49749 [RST] Seq=1 Win=32767 Len=0
229	4.0.165692522	192.168.126.128	10.136.28.34	Portmap	82	V104316 proc-0 Call (Reply In 239)
230	4.0.165630550	192.168.126.128	10.136.28.34	UDP	42	50006 - 22341 Len=0
231	4.0.165894227	192.168.126.128	10.136.28.34	UDP	42	50005 - 36893 Len=0
232	4.0.165921589	192.168.126.128	10.136.28.34	UDP	42	50005 - 1040 Len=0
233	4.0.165950494	192.168.126.128	10.136.28.34	UDP	42	50005 - 682 Len=0
234	4.0.165977366	192.168.126.128	10.136.28.34	UDP	42	50005 - 5560 Len=0
235	4.0.166004287	192.168.126.128	10.136.28.34	UDP	42	50005 - 20146 Len=0
236	4.0.166031368	192.168.126.128	10.136.28.34	UDP	72	50005 - 626 Len=30
237	4.0.166058500	192.168.126.128	10.136.28.34	UDP	42	50005 - 120 Len=0
238	4.0.166085412	192.168.126.128	10.136.28.34	UDP	42	50005 - 19933 Len=0
239	4.0.167961509	10.136.28.34	192.168.126.128	Portmap	74	V104316 proc-0 Reply (Call In 229)
240	4.0.168022546	192.168.126.128	10.136.28.34	ICMP	102	Destination unreachable (Port unreachable)
241	4.1.0.0767090	192.168.126.128	10.136.28.34	RADIUS	62	Access-Request id=0
242	4.1.0.170816354	192.168.126.128	10.136.28.34	UDP	42	50005 - 22695 Len=0
281	5.0.268205129	192.168.126.128	10.136.28.34	UDP	42	50006 - 22695 Len=0

图 5.39: WireShark 抓包结果 2-Nmap UDP 对全部端口扫描

如图5.39所示，前面那条 ICMP 是对另一条源主机向目的主机发送的 Portmap 数据包的回复，另外下面的 ICMP 数据包也是源主机向目的主机发送的，都与条件不符合。因此实际上就是没有收到任何回复的 ICMP 数据包，因此我推测 Nmap 通过了其他方式能够精准确定端口是开放的，而不是过滤的，这也和我们在前面通过查看端口状态的结果一致，再次体现了 Nmap 的强大。

到此所有结果与功能演示完毕，可以看到实验完成度非常高，实现了全部的功能，整体来说较为成功。

### 5.2.7 特殊结果分析

因此综合上面展示与分析，可以看到 Nmap 在 Fin 扫描时候确实没有收到 RST 数据包的回复说明端口关闭，但为什么在确定了目标主机防火墙没有开启的情况下，Nmap 的 SYN(还有 Connect 扫描)的结果会与 Fin 扫描结果不一致。具体而言，是什么导致了 Nmap 的 SYN(还有 Connect 扫描)的结果为端口过滤，而 Fin 扫描结果为端口开启。

在查阅了相关的资料后，我给出下面我的分析：

1. **入侵检测系统 (IDS)**: 尽管防火墙没有启用，网络上可能仍存在入侵检测系统 (IDS) 或入侵防御系统 (IPS)。这些系统通常会监控并过滤异常的 SYN 扫描流量，从而将这些扫描识别为潜在威胁并进行阻止。然而，FIN 扫描由于其隐蔽性，可能不会被这些系统检测到。
2. **SYN 包与 FIN 包处理机制**: 不同的 TCP 扫描方法会触发不同的网络堆栈行为。SYN 扫描发送的 SYN 包是建立连接的一部分，操作系统或中间设备可能会特别处理这些包以防止未授权的访问。FIN 扫描发送的是 FIN 包，这种包通常用于连接终止，不会像 SYN 包那样触发建立连接的机制，因此可能会被忽略或处理不同<sup>1</sup>。
3. **网络设备的过滤规则**: 即使防火墙未开启，网络中的路由器或交换机等设备可能配置了默认的过滤规则。这些设备可能会对 SYN 包进行严格的检查和过滤，而对 FIN 包较为宽松。例如，某些设备可能被配置为丢弃所有未授权的 SYN 包以防止连接建立，但允许 FIN 包通过<sup>2</sup>。
4. **TCP/IP 协议实现的细微差异**: CentOS 7 的 TCP/IP 协议栈可能在处理 SYN 和 FIN 包时存在细微差异。按照 RFC 793 标准，FIN 包不应该触发连接建立，这可能使得 FIN 扫描更难被检测到，从而显示端口为开放状态。而 SYN 包会触发连接建立过程，更容易被操作系统或网络设备检测到并过滤<sup>3</sup>。

**具体分析**: 因此在对 CentOS 7 主机进行扫描时，SYN 扫描显示端口被过滤。这可能是因为网络中的某些设备或操作系统本身配置了对 SYN 包的严格处理规则，即使防火墙未启用。这些设备或系统会将所有未经授权的 SYN 包丢弃或拦截，从而使得扫描结果显示端口被过滤。而 FIN 扫描由于其隐蔽性，这些设备或系统可能未对其进行特殊处理，导致扫描结果显示端口开放。

**结论**: SYN 扫描和 FIN 扫描结果的差异可以由网络设备的默认过滤规则、操作系统的 TCP/IP 协议实现细微差异，以及入侵检测系统的行为等多种因素引起。即使防火墙未启用，这些因素仍然可能导致扫描结果的不同。因此实际结果需要结合多种扫描方式进行综合分析，并考虑具体网络环境反复验证。

## 6 实验遇到的问题及其解决方法

### 6.1 命令行参数校验

- **遇到的问题**: 由于本程序对实验手册中的实验要求有着简单不同的要求，对于 IPv4 地址必须提供，但是端口号可以显式提供也可以交互式提供。因此在用户使用该程序时，显式在命令行参数中输入的参数，若不符合格式要求，可能会出现意料之外的错误。
- **解决的办法**: 我通过在 `utils.cpp` 中定义了 `isValidIPv4` 和 `isValidPort` 对 IPv4 地址和端口都在真正开始扫描前进行了详细的检测验证。若有错误会立即返回。另外程序还有着完整的报错日志，方便确认问题。详细代码详见介绍。

<sup>1</sup>[Source1](#)

<sup>2</sup>[Source2](#)

<sup>3</sup>[Source3](#)

## 6.2 多线程扫描

### 6.2.1 资源共享

- **遇到的问题：**为了提高扫描的效率，本次实验采用了多线程技术来同时扫描多个端口。然而多线程环境下，多个线程可能同时访问和修改共享资源，如果没有妥善的同步机制，就会引发竞态条件，导致程序的行为变得不可预测。此外，不一致的数据会使扫描结果不准确，影响整体的可靠性和稳定性。比如部分 `buffer` 就会出现同时访问的问题。
- **解决的办法：**首先将一些不必要的变量设为局部变量，但是有些必须要共享的变量比如 `TCPConnectThreadNum` 即线程数和 `errorStatusMutex` 即状态错误标志位必须多线程访问。使用 `pthread_mutex_t` 将所有需要同时访问的资源都保护起来。结束使用再释放。

---

```

1 class TCPConnectScanUtil
2 {
3     private:
4     /**
5      * Declaration of the variables
6      */
7     /* TCP Connect Scanning Thread Parameters Struct */
8     static int TCPConnectThreadNum;
9     static pthread_mutex_t TCPConnectThreadNumMutex; // The mutex for the
10    ↳ thread number
11    static int errorStatus;                         // The global error status
12    ↳ for the single thread
13    static pthread_mutex_t errorStatusMutex;        // The mutex for the error
14    ↳ status
15    static ThreadSafeQueue<LogMessage> logQueue;   // The log queue
16    static pthread_mutex_t logQueueMutex;           // The mutex for the log
17    ↳ queue
18    /* blablabla */
19 };
20 // Scan on the specific port in this thread
21 void *TCPConnectScanUtil::Thread_TCPConnectHost(void *param)
22 {
23     /* blablabla */
24     // Push the log message to the queue
25     LogMessage log = {port, logMessage};
26     pthread_mutex_lock(&logQueueMutex);
27     logQueue.push(log);
28     pthread_mutex_unlock(&logQueueMutex);
29
30     // Decrease the number of threads
31 }
```

```

27     pthread_mutex_lock(&TCPConnectThreadNumMutex);
28     TCPConnectThreadNum--;
29     pthread_mutex_unlock(&TCPConnectThreadNumMutex);
30     /* blablabla */
31 }

```

---

### 6.2.2 日志线程

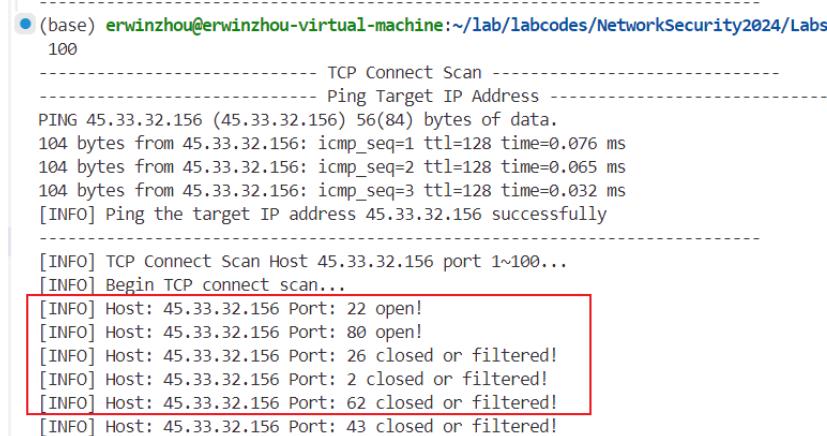
- 遇到的问题:** 另外除了一些变量是比较容易发现的资源之外，另外还有一项资源页会面临着竞争，那就是控制台。实际上由于多线程的端口扫描函数会各自打印各自的消息，因此日志信息也可能会出现混乱的情况，实际上在我实现的早期就发现了这个问题，如下图所示：

```

[INFO] Host: 45.33.32.156 Port: 8 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 48 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 38 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 54 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 58 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 27 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 13 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 35 closed or filtered!
[INFO] Host: [INFO] Host: 45.33.32.156 Port: 36 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 39 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 30 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 44 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 20 closed or filtered!
[INFO] Host: [INFO] Host: 45.33.32.156 Port: 42 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 52 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 24 closed or filtered!

```

图 6.40: 日志顺序混乱 1



```

(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/Labs/Lab04/bin$ sudo ./Scanner -c 45.33.32.156 1
100
----- TCP Connect Scan -----
----- Ping Target IP Address -----
PING 45.33.32.156 (45.33.32.156) 56(84) bytes of data.
104 bytes from 45.33.32.156: icmp_seq=1 ttl=128 time=0.076 ms
104 bytes from 45.33.32.156: icmp_seq=2 ttl=128 time=0.065 ms
104 bytes from 45.33.32.156: icmp_seq=3 ttl=128 time=0.032 ms
[INFO] Ping the target IP address 45.33.32.156 successfully

[INFO] TCP Connect Scan Host 45.33.32.156 port 1~100...
[INFO] Begin TCP connect scan...
[INFO] Host: 45.33.32.156 Port: 22 open!
[INFO] Host: 45.33.32.156 Port: 80 open!
[INFO] Host: 45.33.32.156 Port: 26 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 2 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 62 closed or filtered!
[INFO] Host: 45.33.32.156 Port: 43 closed or filtered!

```

图 6.41: 日志顺序混乱 2

- 解决的办法:** 因此我引入了 `utils.hpp` 中声明的日志线程来解决即

`logProcessingThread(ThreadSafeQueue<LogMessage> &logQueue, int beginPort, int endPort)`。

通过将具体每次多线程产生的日志不再直接打印，而是压入到同样使用线程锁保护好的日志队列中并在日志线程中按照端口号有序输出的方式，解决了该问题。

---

<sup>1</sup> // Declarations of global utility functions utilized in other cpp files

<sup>2</sup>



```

38     // Push the log message to the queue
39     LogMessage log = {port, logMessage};
40     pthread_mutex_lock(&logQueueMutex);
41     logQueue.push(log);
42     pthread_mutex_unlock(&logQueueMutex);
43
44     /* blablabla */
45
46 }

```

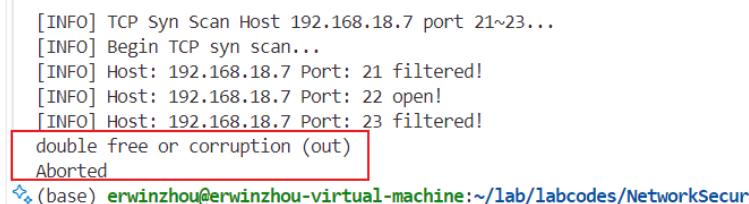
---

### 6.3 管理权限

- **遇到的问题:** 和上次的命令行交互式程序 MD5 不同, 本次实验由于部分扫描方式比如 SYN, FIN 等需要用到原始套接字 SOCK\_RAW, 为了创建这些套接字需要程序有较高的权限, 最开始我在创建 socket 时候一直出错。
- **解决的办法:** 因此为了通过管理员权限运行程序。我在帮助信息中详细指明了运行该程序前面要加上 “sudo”, 这就是为了确保运行的正确性, 赋予程序足够高级的权限。

### 6.4 资源释放

- **遇到的问题:** 最开始我遇到了资源释放错误的问题, 具体而言就是显示错误信息 “double free corruption”。由于本次实验框架较为庞大, 因此在多处涉及到资源管理时候, 难免会出现问题。



```

[INFO] TCP Syn Scan Host 192.168.18.7 port 21~23...
[INFO] Begin TCP syn scan...
[INFO] Host: 192.168.18.7 Port: 21 filtered!
[INFO] Host: 192.168.18.7 Port: 22 open!
[INFO] Host: 192.168.18.7 Port: 23 filtered!
double free or corruption (out)
Aborted
(base) erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecur.

```

图 6.42: 资源释放错误

- **解决的办法:** 通过在线程初始化时候设置线程状态为 PTHREAD\_CREATE\_DETACHED 可有效让线程自动释放资源。同时对 attr 和 struct TCPConnectThreadParam \*p 等变量都小心处理才能够保证资源的释放正确性。

---

```

1   // Set the child thread separated
2   pthread_attr_init(&attr);
3   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

```

---

## 7 实验改善方向

### 7.1 其它扫描方式

实际上除了本次实验要求的端口扫描方式，还有很多其他扫描方式，对于 TCP 而言：

1. 向目标主机发送一个只有 IP 头的 IP 数据报，目标主机将返回 Destination Unreachable 的 ICMP 错误报文。
2. 向目标主机发送一个错误的 IP 数据报，比如，IP 头长度错误。目标主机将返回 Parameter Problem 的 ICMP 错误报文。
3. 当数据包分片，但是却没有给接收端足够的分片，接收端分片组装超时会发送分片组装超时的 ICMP 错误报文。
4. 向目标主机发送一个 IP 数据报，但是协议项是错误的，比如协议项不可用，那么目标将返回 Destination Unreachable 的 ICMP 错误报文。

除此之外还有间接扫描，ACK 扫描，NULL 扫描，FIN+URG+PUSH 扫描等，进一步实现这些方式都能让我网络编程能力得到增强。

### 7.2 深度探索 Nmap 原理

在本次实验的实验分析中，我通过 Wireshark 抓包对比，发现了 Nmap 和端口扫描器的结果有很大的不同，并且有很多机制并不是简单的原理书一样阐述的。它通过重复发送和其它标志位等行为进一步确定端口状态，甚至可以确定端口协议等。为了能够更好地理解这把“瑞士军刀”并写出更有效的端口扫描器，本实验有待进一步探究 Nmap 的源代码，从它的底层逻辑出发，探寻其检测的原理，这不仅有利于加深我对 Nmap 扫描的理解，更会让我对网络安全等基础知识有更深刻的认知。

## 8 实验结论

本次实验是本学期网络安全实验课的最后一次课程作业，在 Linux 平台上实现了多功能的端口扫描器命令行交互程序。除此之外，我还进一步实现了可以指定次数 + 默认次数 + 持续次数的 ping 程序和缺省与给定端口参数的设计。最后我还利用 Wireshark 抓包，将 Nmap 的扫描结果与我的结果进行了详细的分析和总结。总的来说我收获颇丰，具体而言：

1. 本次实验我继续维持着规范的编程架构，通过清晰的架构，让我本次代码具有极高的可读性和可维护性，我也会保持这个习惯，用于我今后的其它项目中。
2. 本次实验我通过对端口扫描器的 ping 程序，TCP Connect Scan, TCP Syn Scan, TCP Fin Scan, UDP Scan 的原理进行复习后，将全部流程进行了实现。在这期间还遇到了线程混乱，资源释放错误等问题，但我凭借耐心一一解决。有的自主创新，并没有完全按照实验参考书的代码和解决思路。这不仅让我对端口扫描器，TCP/IP 协议栈和不同系统的网络端口交互有了更加清晰的认识，并且增强了我的信息安全实践能力。
3. 本次实验在完成了基础的编程要求后，我对自己提出了更高的要求：提高程序的交互性，增强对实验结果的理解。因此我首先配置了另一台 Linux 系统 Centos 作为实验平台，然后试用 Wireshark 抓包仔细分析，将所有端口扫描器的结果都用 Nmap 相同的扫描方式进行对比，遇到了许多不一

样的地方，我通过查阅大量资源并详细分析，提出了自己的看法。这个过程不仅加深了我对 Nmap 的理解，感受到了“瑞士军刀”的强大，并让我锻炼了分析网络抓包交互等网络安全的基础素养，让我印象深刻。

4. 除此之外，我还编写了大型项目所需的 Makefile，对 ping 程序实现中仿照官方 ping 程序进行模仿，让日志信息更加规范和正式。整个程序具有丰富的日志信息，这样的编程习惯让我十分收益。
5. 本次实验后我认真思考与总结，总结了许多实验中遇到的问题，解决问题的思路，我希望我可以在未来对本次实验工作进一步完善。

作为本学期的最后一次实验，让我对端口扫描器等多种扫描方式的实现以及命令行交互式程序都有了深刻的认识。我希望未来能够将网络安全的理论知识继续学透，更好地应用于实践！争取发挥更多的创造性，成为一名有趣的信安人。

## 致谢

这学期的全部实验到此结束了，因为本学期的课程负担不再像前几学期那么繁重，本学期我终于有机会，有时间，有精力把每次实验做好。回忆起这四次实验，我能够将自己的完美主义施展到极致，这为我带来了极大的幸福。能够做一些喜欢的事情，探索喜欢的主题，做出完整而规范的作品，让我本学期对网络安全课程十分满足。

在此，我想向张玉老师表达我最真挚的感谢。张老师在课堂上不仅讲授了丰富的网络安全知识，还结合当今的最新新闻和最新技术，将大模型等前沿技术介绍到我们的课堂中。这些内容不仅丰富了我们的知识面，还让我们能够将理论与实践相结合，增强了我们的实际操作能力。张老师幽默风趣的讲课风格，使得课堂氛围轻松愉快，学习过程变得更加有趣和高效。我从中受益匪浅，不仅提高了专业技能，也培养了对网络安全领域更深的兴趣和热情。

感谢张老师的辛勤付出和耐心指导，也感谢自己在这段时间的努力和坚持，希望在未来的学习和工作中继续保持这种热情和追求卓越的态度，不断探索，勇攀高峰。

## 参考文献