网络空间资产测绘工具设计报告

**目 录**

[网络空间资产测绘工具设计报告 1](#_Toc140699326)

[1、项目目的 2](#_Toc140699327)

[2、项目环境和条件 2](#_Toc140699328)

[3、项目原理 2](#_Toc140699329)

[3.1 主机存活性探测 3](#_Toc140699330)

[3.2 端口开放探测 4](#_Toc140699331)

[3.3 协议与指纹识别 4](#_Toc140699332)

[3.4 设备识别 7](#_Toc140699333)

[3.5 蜜罐识别 7](#_Toc140699334)

[4、项目内容 8](#_Toc140699335)

[4.1主机存活性探测 8](#_Toc140699336)

[4.2 端口开放探测 10](#_Toc140699337)

[4.3 协议与指纹识别 11](#_Toc140699338)

[4.4 设备识别 21](#_Toc140699339)

[4.5 蜜罐识别 22](#_Toc140699340)

[4.5.1 Glastopf蜜罐： 22](#_Toc140699341)

[4.5.2 Kippo蜜罐： 22](#_Toc140699342)

[4.5.3 Hfish蜜罐： 22](#_Toc140699343)

[5 项目总结与心得体会 23](#_Toc140699344)

# 1、项目目的

针对Build环节中的要求，尽可能的对网络空间测绘的结果有更高的精确度和回归率。

# 2、项目环境和条件

Linux服务器下，安装Python3及nmap

# 3、项目原理

**收集资料**

题目的要求是写一个网络空间资产测绘工具，要对对互联网上资产与服务的测绘、全端口测绘、深度交互式测绘和蜜罐发现等网络空间资产测绘能力。首先想到的就是nmap，于是搜索了一下nmap的源码分析

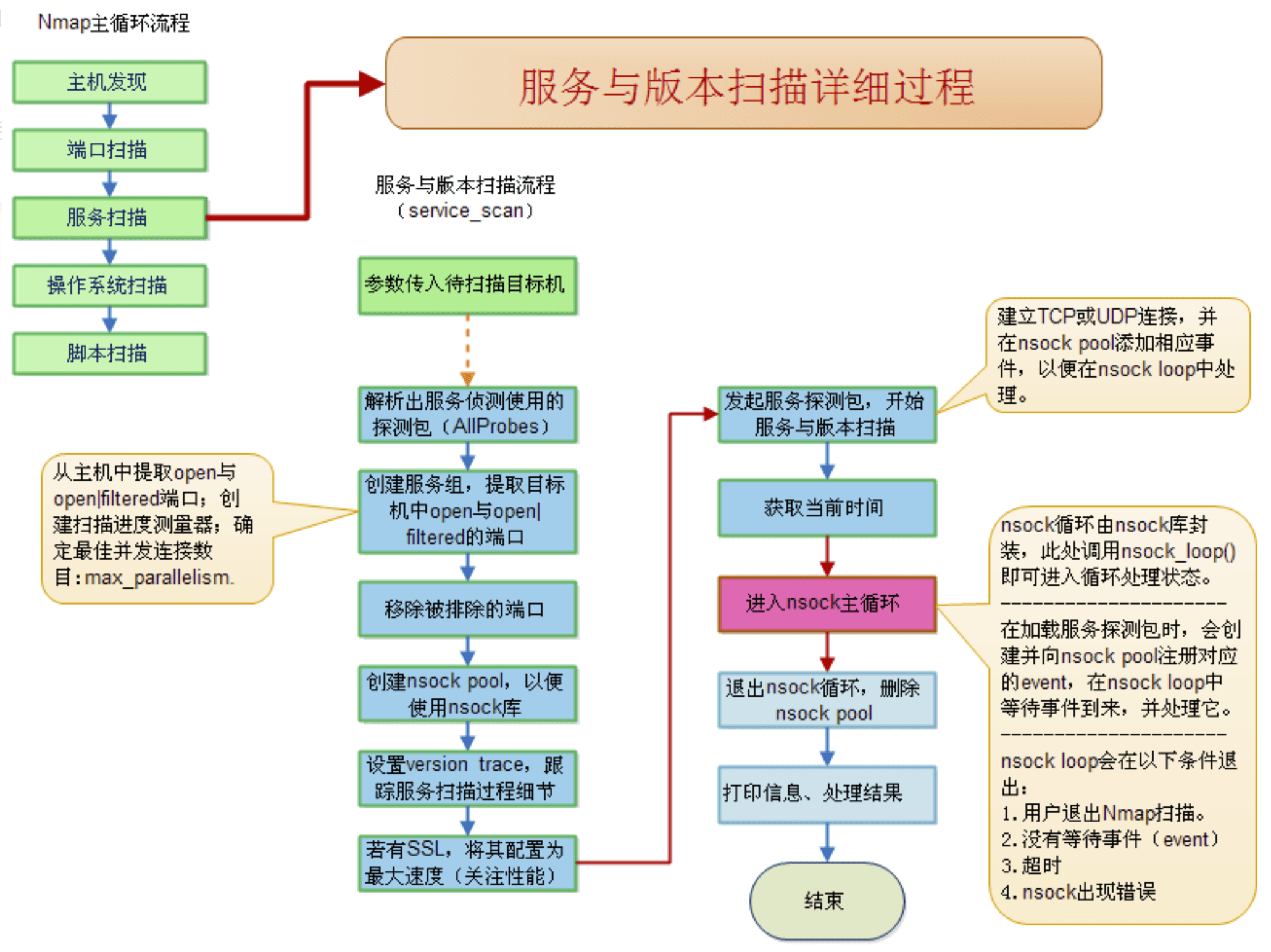


图3-1 nmap程序流程

根据这张图nmap主要的扫描流程是存活主机发现、开放端口扫描、服务扫描，然后再加上题目要求的设备识别和蜜罐识别，那么我就打算把这个测绘工具分成五个部分。

（1）主机存活性探测

（2）端口开放探测

（3）协议与指纹识别

（4）设备识别

（5）蜜罐识别

## 3.1 主机存活性探测

本项目中存活主机探测是使用的ping进行的探测，并使用多线程的方式进行发包，只发送两个数据包，检测返回包中ms关键字，有就说明这个ping的包是正常返回的，就认为这个主机是存活的主机。

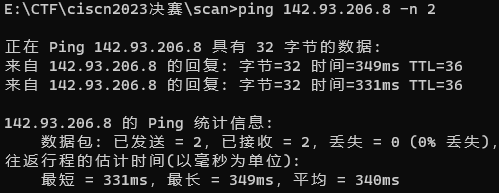


图3-2 ping进行探测的返回包

使用ping进行主机存活探测有以下优势：

快速：Ping命令发送ICMP回显请求（Echo Request）到目标主机，并等待目标主机返回ICMP回显应答（Echo Reply）。这个过程非常快速，通常在毫秒级别完成，因此可以迅速判断主机是否存活。

简单：Ping命令是操作系统自带的命令之一，使用方便且无需额外安装软件。它的简单性使得主机存活探测变得容易，无论是在命令行还是在脚本中都能轻松使用。

广泛支持：Ping命令在多种操作系统和网络设备上都得到广泛支持，包括Windows、Linux、macOS等。这意味着可以在不同平台上进行主机存活探测，使其具有普适性。

实时性：Ping命令可以提供即时的结果。通过检测目标主机是否响应ICMP回显请求，可以快速确定主机的存活状态，而无需等待较长时间的超时。

可定制性：Ping命令还可以通过设置参数来控制探测的方式和结果的显示。可以设置发送的ICMP包数量、超时时间、数据包大小等参数，以满足不同的需求。

低资源消耗：Ping命令的资源消耗较低，不会对目标主机造成太大的负担。它通常发送小型ICMP数据包，并接收简单的ICMP应答，对网络和主机的资源消耗较小。

## 3.2 端口开放探测

经过多种尝试，发现使用python3的socket的库对端口进行连接效率更高，由于扫描1-65535端口所需消耗的资源过大，于是先对所给的网段的所有端口进行了一次探测，然后记录下出现过的端口，下次扫描时就重点扫描这些出现过的端口，牺牲了一定的准确度，但是大大节省了端口扫描的时间。

## 3.3 协议与指纹识别

这里直接使用的python的socket库对各个端口进行连接

FTP：FTP需要用socket进行端口连接得到的返回信息中就有FTP的版本信息，做一个正则匹配就可以得到FTP的版本信息。

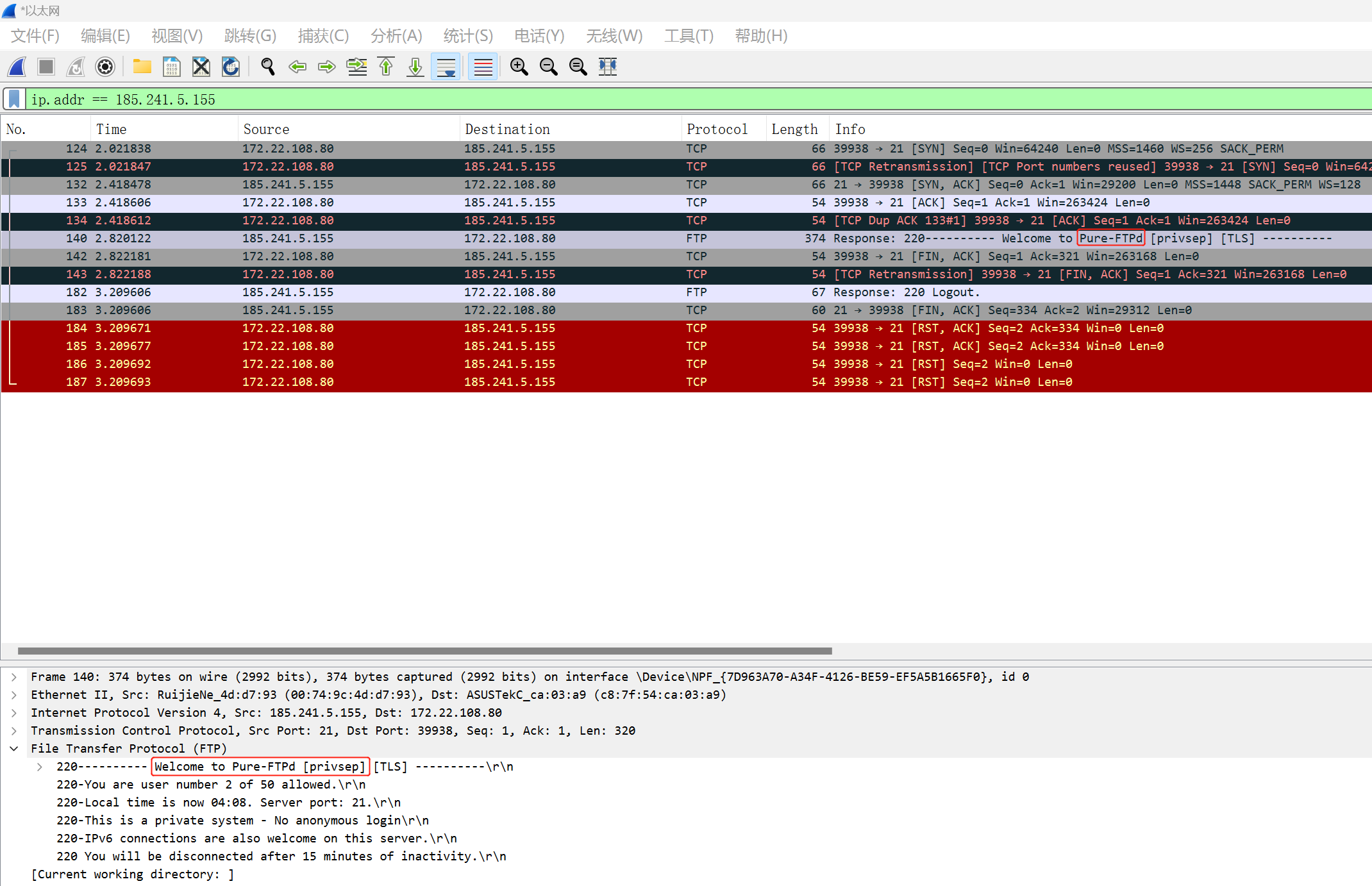


图3-3 FTP返回的版本信息

SSH：SSH需要socket进行端口连接就可以获取到版本信息。

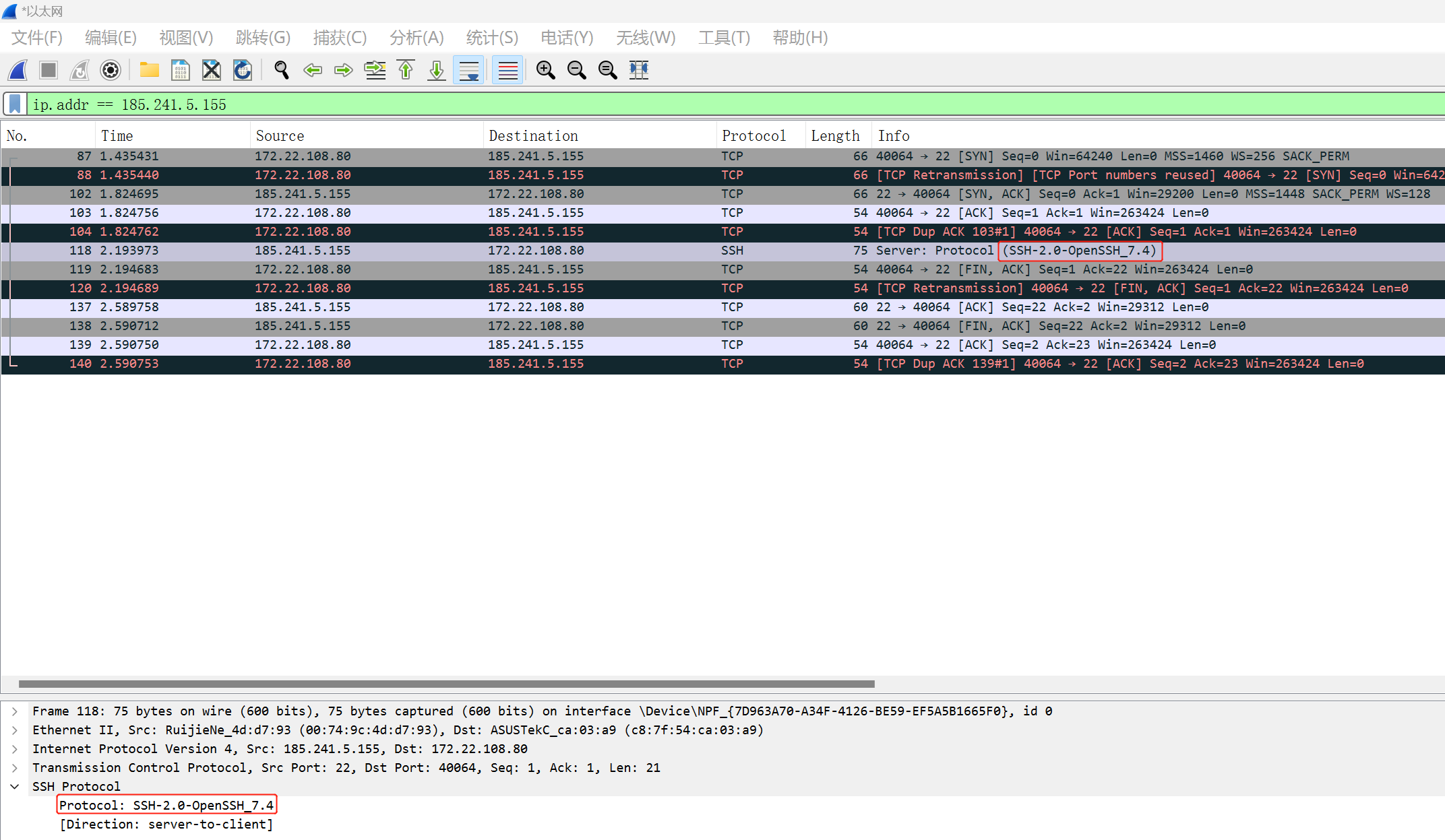


图3-4 SSH返回的版本信息

Telnet：23端口开放就认为是Telnet服务。

HTTP：HTTP这里就有一些复杂，使用的是requests包，通过head请求，然后再通过解析响应包中的Server字段得到版本信息，如果Server字段是一些不在公告所给列表中的组件，那么就改用get请求，请求具体页面的404页面，然后匹配显示的Web服务器。



图3-5 Server字段中网站信息

HTTPS：HTTPS协议原本是打算通过特殊的访问HTTPS的结构进行访问的，但经过实测后发现大量的网站HTTPS证书并没有配置好，只是一个开在443端口的HTTP服务，所以HTTPS就沿用了HTTP的方法进行判断。

Mysql：用socket进行连接，对于返回的信息匹配出版本号即可。

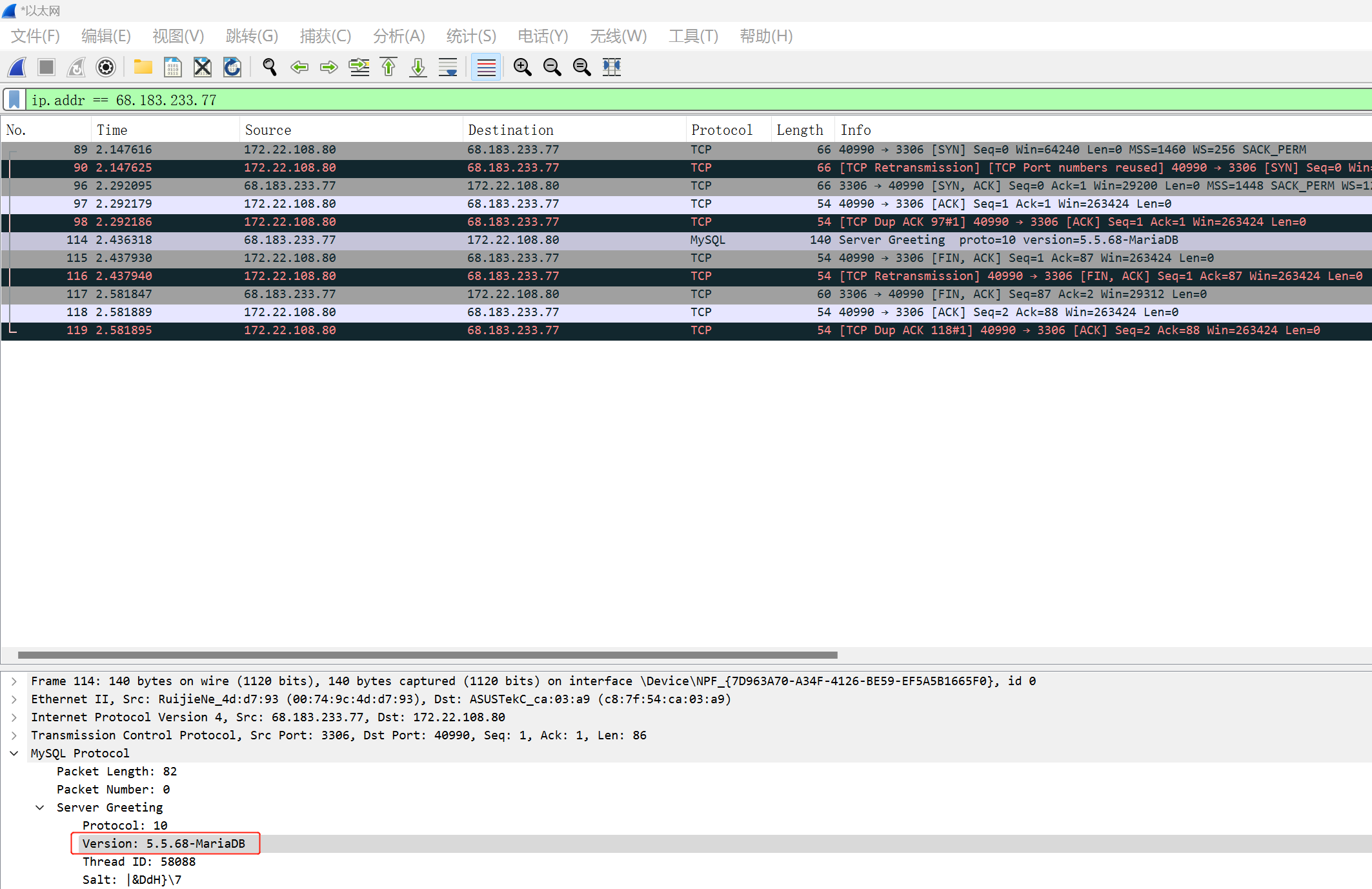


图3-6 Mysql返回版本信息

ampq：ampq协议暂时认为是5672端口。

rabbitmq：rabbitmq协议暂时认为是15672。

剩下的一些知名端口，用socket库直接匹配协议

一些开放在高位的端口，其实本身就是用上面已经写好的解析方式对每个高位端口进行尝试，尝试成功那就是对应的协议，尝试失败就是未知协议，由于比较麻烦，就借助nmap来进行扫描判别，得到高位端口的协议和版本号。

网站指纹的测绘通常从以下几个方面考虑、页面源码、header、图标hash、静态文件内容和静态文件哈希

1首页源码

匹配首页源码的相似度，考虑对首页源码做md5比较是否一致。

2首页header

匹配首页响应Header的内容是否-致，例如Header中的server: LPS-0.1，一致则输出何种网页的指纹

3图标hash 对具有网站特征的图标做hash并分类，匹配是否一致。

#注:所有CMS使用的同类图片需分别计算hash并归类，例如tomcat的png和icon等文件

4静态文件内容

对首页源码做md5比较是否一致

5静态文件hash匹配指定URL下的文件Hash，方法同3.3，但通常感觉不如3.3，因为文件会做改动，可能在比赛环境中更合适，需要见到几个静态代码对比一下

6召回率

由于比赛文档中提出指纹经过特殊设计，那么就考虑是属于二次开发的范围。在二次开发的场景下，尽可能多尝试使用图标hash、CMS中body的标识型符号等标识性内容，考虑独有的islcss文件或者特殊独有的color数直，例Discuz系统的body="<meta name="generator" content="Discuz!”"body="name="srhlocality"

以上方法取结果的并集，能够提高极大的召回率

## 3.4 设备识别

遍历请求所有HTTP的端口，在页面中有对应的设备指纹，那就判定为对应的设备。

例如，HTTP请求后，发现该网站是pfsense，那么就判定该ip的设备是[firewall/pfsense]

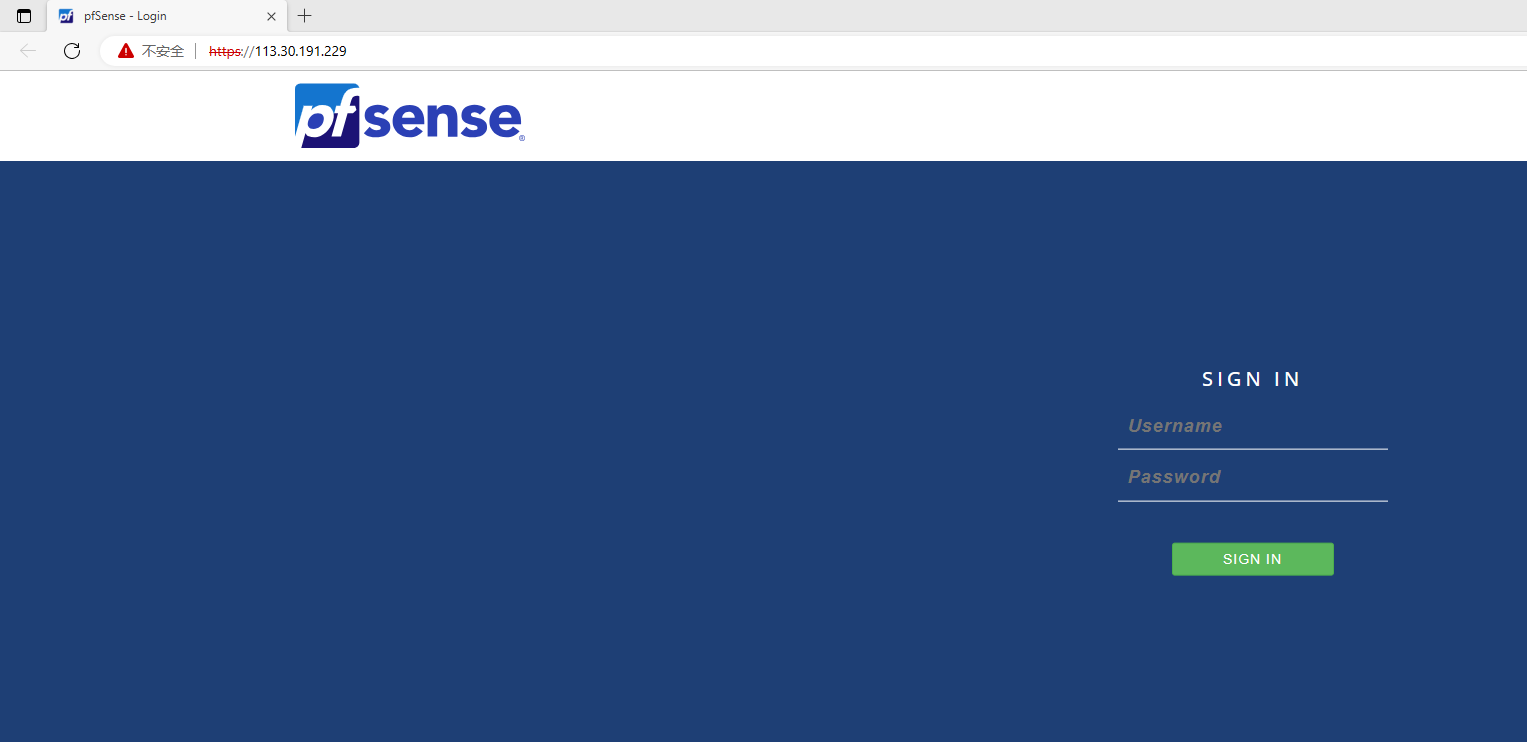


图3-7 识别pfsense指纹

## 3.5 蜜罐识别

在组委会公开的蜜罐列表中，出现了三种蜜罐，分别是glastopf、Kippo、HFish，我们在本地搭建了这几种蜜罐之后，与他们进行交互，并提取了蜜罐的规则特征作为蜜罐识别依据。

# 4、项目内容

## 4.1主机存活性探测

ping(ip, ip\_alive, sc): 这个函数用于执行ping命令，并将结果存储在ip\_alive字典中。ip是要ping的目标IP地址，ip\_alive是一个用于存储存活主机的字典，键为IP地址，值为"1"表示存活，sc是一个信号量（Semaphore）用于控制并发线程数量。

scan\_alive\_ip(ip): 这个函数是多线程探测主机是否存活的入口函数。它会遍历一个IP地址范围（例如：192.168.1.x）并使用多线程方式调用ping函数来检测每个IP地址的存活情况。最后，它会返回一个字典ip\_alive，其中记录了该网段中存活主机的IP地址和状态。

|  |
| --- |
| # 1 探测存活主机  # 执行ping命令并返回结果  def ping(ip, ip\_alive, sc):  # 根据操作系统选择探测命令  system\_operate = getsystem()  # Windows  if system\_operate == "Windows":  pings=subprocess.Popen('ping -n 2 %s' % ip , shell=True, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE, encoding='gbk') #这时windows，linux为ping -c 2  result=pings.stdout.read()  # Linux  if system\_operate == "Linux":  pings=subprocess.Popen('ping -c 2 %s' % ip , shell=True, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE, encoding='utf-8') #这时windows，linux为ping -c 2  result, \_ = pings.communicate()  if 'ms' in result: # 正确的结果里面有ms 就是时间的单位  # print(ip,'open')  ip\_alive[ip] = "1"  # 多线程探测主机是否存活，用的ping探测，如遇到禁ping主机无法  # 返回ip 与 是否存活 的字典  def scan\_alive\_ip(ip):  ip\_alive = {} # 记录一个网段主机是否存活  threads\_ip = [] # 为了存线程  semaphore=threading.Semaphore(20) # 最多允许同时执行的线程数  for i in range(1,256):  t=threading.Thread(target=ping, args=(ip+str(i), ip\_alive, semaphore))  t.start()  threads\_ip.append(t)  for t in threads\_ip: # 等所有进程结束后  t.join()    print(ip\_alive) # 输出这个网段存活主机的结果，格式为{ip:1/0} 后续将1/0改为具体端口和服务  # print(len(ip\_alive.keys()))  return ip\_alive # ip\_alive json数据格式例：{'16.163.13.23':1,...}  # 由于跨平台，每次换平台添注释删注释麻烦  def getsystem():  # 获取当前操作系统的名称  system = platform.system()  return system |

## 4.2 端口开放探测

scan\_ports(target\_ip): 这个函数是入口函数，用于扫描指定IP地址的端口是否开放。它会遍历scan\_ports\_from\_file中指定的端口范围（在代码中未给出，需要根据具体情况查看），然后为每个端口创建一个线程，并调用scan\_port函数进行端口扫描。扫描结果会被记录在open\_ports列表中，并在完成所有线程的执行后输出IP地址及其开放的端口。

scan\_port(port, target\_ip): 这个函数是用于扫描指定端口的实际函数。它会创建一个TCP套接字，然后尝试与指定的IP地址和端口建立连接。如果连接成功（sock.connect\_ex((target\_ip, port))返回0），则说明端口是开放的，将该端口加入到open\_ports列表中。若连接超时或被拒绝，将忽略异常，不进行处理。

|  |
| --- |
| # 2 探测端口是否开放  # 对一个端口进行一次syn扫描  # 输入一个ip与一个端口  def scan\_ports(target\_ip):  # 创建线程并执行端口扫描  threads = []  # 统计所有开放的端口  global open\_ports  open\_ports = []  print("scan IP:%s" % target\_ip)  for port in scan\_ports\_from\_file: # 扫描端口范围从1到10001  t = threading.Thread(target=scan\_port, args=(port, target\_ip))  t.start()  threads.append(t)  # 等待所有线程执行完毕  for t in threads:  t.join()  print("IP %s Open ports:%s" % (target\_ip, open\_ports))  return (target\_ip, open\_ports)  # 扫描指定端口的函数  def scan\_port(port, target\_ip):  sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  sock.settimeout(2)    try:  result = sock.connect\_ex((target\_ip, port))  if result == 0:  with lock:  print("Port", port, "is open")  open\_ports.append(port)    except (socket.timeout, ConnectionRefusedError):  pass    finally:  sock.close() |

## 4.3 协议与指纹识别

主要分为三部分，第一部分是端口的分流，第二部分是根据对应的端口用socket进行连接根据对应的返回进行解析，第三部分是使用nmap对难以鉴别和处理的端口进行扫描。

首先常用的知名端口，以及上述提到的好处理版本号的协议用自己写的函数进行处理，并用remain\_ports记录没有被扫描的端口，scan\_port返回已经扫描好的记录好port,protocol,service\_app的内容。

|  |
| --- |
| # 3 探测开放端口具体服务  # 对一个端口进行一次syn+ack扫描  # 输入一个ip与一个端口  def scan\_port\_with\_version(target\_ip, open\_ports):    # 存放已经扫过的端口  scanned\_ports = []  # scan\_port = {target\_ip: {"service":[]}}  # 一个ip所有端口的服务具体信息  scan\_port = {"services":[]}  for target\_port in open\_ports:  if target\_port == 21: # 不el 就会认为最后一个if前面是一个新的if然后最后又多输出一遍  ftp\_version = get\_ftp\_version(target\_ip, target\_port)  scan\_port["services"].append(ftp\_version)  # 扫描过的端口记录  scanned\_ports.append(target\_port)    # 直接探测 ssh 具体版本 还是socket牛逼  elif target\_port == 22:  ssh\_service = scan\_port\_ssh\_version(target\_ip, target\_port)  # scan\_port[target\_ip]["service"].append[ssh\_version]  scan\_port["services"].append(ssh\_service)  # 扫描过的端口记录  scanned\_ports.append(target\_port)  # 23 端口暂时直接赋telnet  elif target\_port == 23:  telnet\_version = telnet\_with\_version(target\_ip, target\_port)  scan\_port["services"].append(telnet\_version)  # 扫描过的端口记录  scanned\_ports.append(target\_port)  # 80 端口暂时直接赋http  elif target\_port == 80:  http\_version = http\_with\_version(target\_ip, target\_port)    # 若有设备指纹  if http\_version["service\_app"] == "cisco":  scan\_port["deviceinfo"] = deviceinfo\_with\_version(target\_ip, target\_port)  http\_version["service\_app"] = None # 把app职位None  # 这里pfsense直接识别  elif http\_version["service\_app"] == "pfsense":  scan\_port["deviceinfo"] = ["firewall/pfsense"]  http\_version["service\_app"] = ["nginx/N"] # 暂时看到pfsense是nginx的不知道有没有别的  scan\_port["services"].append(http\_version)  # 扫描过的端口记录  scanned\_ports.append(target\_port)  # 443 端口暂时直接赋https  elif target\_port == 443:  https\_version = https\_with\_version(target\_ip, target\_port)  scan\_port["services"].append(https\_version)  # 扫描过的端口记录  scanned\_ports.append(target\_port)  # 3306端口开放直接认为是mysql  elif target\_port == 3306:  mysql\_version = mysql\_with\_version(target\_ip, target\_port)  scan\_port["services"].append(mysql\_version)    # 扫描过的端口记录  scanned\_ports.append(target\_port)  # 5672端口开放认为是ampq  elif target\_port == 5672:  mysql\_version = ampq\_with\_version(target\_ip, target\_port)  scan\_port["services"].append(mysql\_version)    # 扫描过的端口记录  scanned\_ports.append(target\_port)  # 6379端口开放直接认为是redis  elif target\_port == 6379:  redis\_version = redis\_with\_version(target\_ip, target\_port)  scan\_port["services"].append(redis\_version)  # 扫描过的端口记录  scanned\_ports.append(target\_port)  elif target\_port == 15672:  rabbitmq\_version = rabbitMQ\_with\_version(target\_ip, target\_port)  scan\_port["services"].append(rabbitmq\_version)  # 扫描过的端口记录  scanned\_ports.append(target\_port)  # 一些常用端口且不要求版本号的，直接用socket的获取协议的库  elif target\_port == 25 or target\_port == 110 or target\_port == 111 or target\_port == 143 or target\_port == 445 or target\_port == 554:  common\_ports\_version = common\_ports\_with\_version(target\_ip, target\_port)  scan\_port["services"].append(common\_ports\_version)  # 扫描过的端口记录  scanned\_ports.append(target\_port)  # 剩余端口，准备做更详细的扫描  remain\_ports = [x for x in open\_ports if x not in scanned\_ports]  return scan\_port, remain\_ports |

其次就是每个端口具体的处理方法，这里举SSH和HTTP为例。

首先是SSH协议：

scan\_port\_ssh\_version(target\_ip, target\_port): 这是入口函数，用于获取SSH服务的版本信息。它接收目标主机的IP地址(target\_ip)和SSH服务的端口号(target\_port)作为参数。然后创建一个TCP套接字，尝试与目标主机建立连接，如果连接成功，将接收SSH服务器发送的版本数据。

在连接成功后，通过sock.recv(1024)接收SSH服务器的版本数据，然后对接收到的版本数据进行处理，将其存储在version变量中。接着，将版本信息进行解析，将其拆分为服务版本号、操作系统或其他信息，并将结果存储在result\_dict字典中。

如果解析出版本信息成功，将包含端口、协议、服务版本号等信息的result\_dict返回。否则，将返回一个包含端口、协议、服务版本号为None的result\_dict。

|  |
| --- |
| # 获取ssh版本信息 22  def scan\_port\_ssh\_version(target\_ip, target\_port):  # 创建 TCP 套接字  sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  try:  # 设置连接超时时间为2秒  sock.settimeout(2)  # 尝试连接到目标主机  sock.connect((target\_ip, target\_port))  # 接收 SSH 服务器版本号  version\_data = sock.recv(1024)  # version = version\_data.decode().strip().split(" ")[0] 不确定ssh版本返回是否都是一种形式  try:  version = version\_data.decode().strip()  print("SSH version:", version)  version = version.split(" ")  result\_dict = {  "port": target\_port,  "protocol": "ssh",  "service\_app": version  }  sock.close()  return result\_dict    except:  result\_dict = {  "port": target\_port,  "protocol": "ssh",  "service\_app": None  }  sock.close()  return result\_dict    except (socket.timeout, ConnectionRefusedError):  print("Unable to connect or SSH is not running on the target host.")  result\_dict = {  "port": target\_port,  "protocol": "ssh",  "service\_app": None  }  sock.close()  return result\_dict  finally:  # 关闭套接字  sock.close() |

其次是HTTP协议：

首先开头这里是为了接入开了443端口但没有配置好HTTPS证书的端口，就用head请求进行请求，获取相应包中的Server字段并进行处理，如果出现一些特殊的Server字段，则会用get请求进行访问，获取页面内容的指纹信息。

|  |
| --- |
| # http暂时直接返回http 80  def http\_with\_version(target\_ip, target\_port):  # 经过测试发现大部分https证书都不对，用https全是报错，就加个443端口就行了  if target\_port == 443:  url = "http://" + target\_ip + ":443"  protocol = "https"  else:  # 请求变成http://ip  url = "http://" + target\_ip  protocol = "http"  try:  # 发送 HEAD 请求到指定 URL  response = requests.head(url, timeout=5)  # 提取服务器软件版本  version = response.headers.get('Server')  # 这里获取不到的话直接返回吧  if version == None:  result\_dict = {  "port": target\_port,  "protocol": protocol,  "service\_app": version  }  return result\_dict  print(version)  version = version.strip(" ")  # 这里有设备指纹，直接跳到设备  if "cisco" in version:  result\_dict = {  "port": target\_port,  "protocol": protocol,  "service\_app": "cisco"  }  return result\_dict  # 这里开始对返回的服务信息进行处理  if "(" in version and ")" in version:  if "OpenSSL" in version:  if "PHP" in version:  version = version.split(" ")  try:  web\_version = version[0] # web服务器版本  operation\_version = version[1].strip("()")+"/N" # 操作系统版本  openssl\_version = version[2]  script\_version = version[3]  result\_dict = {  "port": target\_port,  "protocol": protocol,  "service\_app": [web\_version, operation\_version, openssl\_version, script\_version]  }  except:  result\_dict = {  "port": target\_port,  "protocol": protocol,  "service\_app": version  }  else:  … 其他指纹识别  # 不给版本号的nginx  elif "nginx" in version:  res = requests.get(url, timeout=5)  if "pfsense" in res.text:  result\_dict = {  "port": target\_port,  "protocol": protocol,  "service\_app": ["pfsense"]  }  else:  result\_dict = {  "port": target\_port,  "protocol": protocol,  "service\_app": [version+"/N"]  }  elif:  … 其他指纹识别  print(result\_dict)  return result\_dict  except requests.exceptions.RequestException:  # 如果连接超时，或者连接拒绝暂时都认为是http  result\_dict = {  "port": target\_port,  "protocol": protocol,  "service\_app": None  }    return result\_dict |

## 4.4 设备识别

请求http协议的端口，匹配到对应的设备就返回

|  |
| --- |
| # 4 设备指纹  def deviceinfo\_with\_version(target\_ip, target\_port):  # 通过http来获得设备信息  # 请求变成http://ip  url = "http://" + target\_ip    try:  # 发送 HEAD 请求到指定 URL  response = requests.head(url, timeout=5)  # 提取服务器软件版本  version = response.headers.get('Server')  # 这里获取不到的话直接返回吧  if version == None:  return None    print(version)  version = version.strip(" ")  # 这里有设备指纹，直接跳到设备  if "cisco" in version:  return [version]  except requests.exceptions.RequestException:  return None |

## 4.5 蜜罐识别

### 4.5.1 Glastopf蜜罐：

glastopf是一个Web蜜罐，它通过模拟Tomcat登录页面来吸引攻击者，它的特征是页面中存在Login Form、My Resource字段，所以只需要访问一下主页，判断是否包含这几个字段就能初步判断是否是一个glastopf蜜罐。

### 4.5.2 Kippo蜜罐：

Kippo是一个中交互的ssh蜜罐，它也是Cowire的原型，经过我们与它的交互发现，Kippo的特征是默认口令为root:123456，而且登陆上去，使用hostname命令得到的用户名为svr03，根据这两点就能够判断一个蜜罐是否为Kippo蜜罐。

### 4.5.3 Hfish蜜罐：

Hfish具有许多蜜罐，其中包含各种模板的Web蜜罐、SSH蜜罐、Mysql蜜罐、Redis蜜罐、Telnet蜜罐，根据分析，他们都具有以下特征：

* + Web蜜罐在访问到当前网站不含有的页面时，页面中只含有404 page not found字样，做一个强匹配，即很大概率可识别到Hfish Web蜜罐，在低版本的HFish中，Web蜜罐模板中也会含有x.js这一个特殊的文件，以此作为第二判断条件。
  + SSH蜜罐与Kippo不同，它的默认账户为root:root，以此作为判断条件判断Hfish SSH蜜罐。
  + Mysql蜜罐以默认登陆账户root:root为识别条件。
  + Redis蜜罐以默认登陆账户root:123456作为识别条件。
  + Telnet蜜罐以空口令连接作为识别条件。

# 5 项目总结与心得体会

这一次的项目开发让本团队的成员详细了解到了网络测绘工具实现的原理，以及对各种常用协议如SSH、FTP、HTTP等获取其版本号及指纹的方法及解析技巧。并且在开发过程中我也获得了如下宝贵的心得体会：

需求明确：在开始开发之前，要确保对测绘工具的需求进行充分的了解和明确。明确工具的功能、目标对象、输出结果等，这有助于指导开发过程，避免走弯路。

使用合适的工具和库：网络空间测绘工具通常需要处理网络通信、数据处理、数据分析等复杂任务，使用合适的工具和库可以大大提高开发效率。Python的第三方库如Socket、Requests、BeautifulSoup等为开发者提供了丰富的功能和工具，帮助快速实现功能。

考虑并发和扩展性：网络测绘工具通常需要处理大量的目标，考虑并发和扩展性是非常重要的。使用多线程、异步编程等技术可以提高扫描效率，而且在设计时要考虑工具的可扩展性，方便后续增加新功能。

良好的文档和注释：为了方便后续维护和使用，要为代码编写良好的注释和文档。注释能够帮助他人理解代码逻辑，文档能够描述工具的使用方法和参数说明。

不断学习和改进：网络空间测绘是一个不断变化和发展的领域，要保持持续学习和改进。关注新的技术和工具，参与安全社区交流，从中学习和获取新的思路和解决方案。

严谨的测试：网络测绘工具在实际应用中可能面临各种复杂的网络环境和目标情况，要进行充分的测试，确保工具的稳定性和准确性。